TACHYON Forth GLOSSARY

The purpose of this glossary is to provide a sorted list of words, their effects, and a short description. It is recommended to go to the relevant source code to glean more information if necessary.

FORMAT OF GLOSSARY

NAME STACK NOTATION DESCRIPTION
ALIAS(ES)

Stack notation shows input parameters with the rightmost being the top of the stack and the output parameters are separated by a --

ALIAS(ES) point to the same code but are made available so that those with traditional Forth or Spin experience can be familiar with them although some are favored for clarity and readability especially in various fonts.

NOTES:

The data stack is 4 levels deep in the cog and then implemented as a non-addressable LIFO stack in cog memory. Tachyon words are optimized for these four fixed cog registers and to encourage efficient stack use no messy PICK and ROLL words are implemented. There are many words that also avoid pushing and popping the stack as this slows execution speed too. Try to factor words so that they use four or less parameters.

STACK WORDS

NAME	*STACK*	**xx** *TY*	*DESCRIPTION*
DUP	(a -- a a)	C	Duplicate top item on stack (push)
OVER	(a b -- a b a)	C	Copy 2nd stack location to first (push)
DROP	(a --)	C	Drop top item off the stack (pop)
2DROP	(a b --)	C	Drop top 2 items off the stack (pop)
3DROP	(a b c --)	C	Drop 3 top stack items
SWAP	(a b -- b a)	C	Swap top 2 stack items
ROT	(a b c -- b c a)	C	Move 3rd item to 1st, 1st to 2nd, 2nd to 3rd
-ROT	(a b c -- c a b)	H	Reverse rotate (equiv. to ROT ROT)
NIP	(a b -- b)	C	Drop 2nd stack item (pop)
?DUP	(a -- a a)	C	dup if a <>0, else (a -- a)
3RD	(a b c d -- a b c d b)	C	Copy third stack item (push)
4TH	(a b c d -- a b c d a)	C	Copy fourth stack item (push)
2DUP	(a b -- a b a b)	H	Duplicate the top two stack items (equiv. to OVER OVER) (push)
2OVER	(a b c d -- a b c d a b)	H	Duplicate the next two stack items (as if it is a double number) (push)
!SP	(? --)	H	Init stack pointer, clear the stack
DEPTH	(-- depth)	H	Return with current depth of data stack (but does not include depth itself) (push)
LP!	(a --)	H	Set loop stack memory - each cog that uses FOR NEXT needs room for 8 longs or more

LOGICAL

Some logical operations include a built-in parameter to avoid slow push/pop operations such as 8<< rather than 8 << (0.4us vs 2.4us)

INVERT	(a -- b)	C	Bitwise inversion - all bits are flipped (i.e. $FFFFFFF5 -> $0A)
AND	(a b -- c)	C	c = a AND b
ANDN	(a b -- c)	C	c = a AND NOT b ($DEADBEEF $FF ANDN .LONG DEAD.BE00 ok)
OR	(a b -- c)	C	c = a OR b ($123400 $56 OR .LONG 0012.3456 ok)
XOR	(a b -- c)	C	c = a XOR b ($123456 $FF XOR .LONG 0012.34A9 ok)
ROL	(a cnt -- c)	C	Rotate a left with b31 rotating into b0 for cnt ($12345678 8 ROL .LONG 3456.7812 ok)
ROR	(a cnt -- c)	C	Rotate right bit b0 rotating into b31 for cnt ($DEADBEEF 8 ROR .LONG EFDE.ADBE ok)
SHR	(a cnt -- c)	C	Shift right all bits by cnt
>>			
8>>	(a -- b)	C	Fast 8-bit shift right - avoids slow push and pop of literal (i.e. $12345678 -> $00123456)
16>>	(a -- b)	C2	Fast 16-bit shift right (made up of two 8>>)
SHL	(a cnt -- c)	C	Shift left all bits by cnt
<<			
8<<	(a -- b)	C	Fast 8-bit shift left - avoids slow push and pop of literal
9<<	(a -- b)	C2	Fast 9-bit shift left - useful for scaling by 512 as in sector sizes etc
16<<	(a -- b)	C2	Fast 16-bit shift left (equiv to 8<< 8<<)
2/	(a -- b)	C	Shift right one bit (divide by two unsigned)
2*	(a -- b)	C	Shift left one bit (multiply by two unsigned)
REV	(n1 bits -- n2)	C	Reverse LSBs of n1 and zero-extend
MASK	(bit -- mask)	C	Convert 5-bit number to a mask over 32-bits - mask=0 if bit>31
|<			
>N	(n -- nib)	C	Mask off a nibble ($0F AND)
>B	(n -- byte)	C	Mask off a byte ($FF AND)
9BITS	(n -- 9bits)	C	Mask off 9-bits ($1FF AND)
>|	(mask -- bit)	X	Convert mask to bit position of first lsb that is set
NOP		C	No operation (0.4 us)

COMPARISON

0=	(val -- flg)	C	Compare n to zero and return with boolean flag
NOT			
=	(a b -- flg)	C	Compare a with b
>	(a b -- flg)	C	If a > b then flg = true
0<>	(n -- flg)	C2	Return with flag indicating if n <> 0 (equiva. to 0= NOT)
<>	(n1 n2 -- flg)	C2	Return with flag indicating if n1 <> n2 (equiv. to = NOT)
0<	(val -- flg)	H	If val is less than zero (negative) then return with true flag
<	(a b -- flg)	H	If a is less than b then return with true flag
U<	(a b -- flg)	H	If a is unsigned less than b then return with true flag
WITHIN	(val min max -- flg)	H	Return with flag if val is within min and max (inclusive, not ANSI)
<=		X	
=>		X	
U>		X	

MEMORY

@	(adr -- long)	C	Fetch a long from hub memory (0.4us)
!	(long adr --)	C	Store the long in hub memory (2.2us) (pops)
W@	(adr -- word)	C	Fetch a word from hub memory
W!	(word adr --)	C	Store word to word at adr
C@	(adr -- byte)	C	Fetch a byte from hub memory
C!	(byte adr --)	C	store byte to hub memory
+!	(long adr --)	C	Add long to long in hub memory
W+!	(word adr --)	C	Add word to word at adr
C+!	(byte adr --)	C	add byte to hub memory
C@++	(adr -- adr+1 char)	C	Fetch a byte from hub memory and maintain and increment the address
SET	(mask addr --)	H	Set the bits in the byte at addr
CLR	(mask addr --)	H	Clear the bits in the byte at addr
SET?	(mask addr -- flg)	H	Test the bits in the byte at addr and return with state
CMOVE	(src dst cnt --)	C2	CMOVE bytes from src to dst by cnt bytes (13.11ms for 32k)
<CMOVE	(src dst cnt --)	H	Reverse MOVE bytes starting from end of src to end of dst by cnt bytes
ERASE	(adr cnt --)	H	ERASE memory (to 0) from adr for cnt bytes
FILL	(adr cnt ch --)	H	FILL memory from adr for cnt bytes with ch
CELLS	(n -- bytes)	C2	Return with the equivalent number of bytes for this processor's word length (32-bits).
C~~	(adr --)	H	Set the byte in hub memory to all ones
C~	(adr --)	H	Clear the byte in hub memory to zeros
W~~	(adr --)	H	Set the word in hub memory to all ones
W~	(adr --)	H	Clear the word in hub memory to zeros
~~	(adr --)	H	Set the long in hub memory to all ones
~	(adr --)	H	Set the long in hub memory to zeros
C--	(adr --)	H	Decrement the byte in hub memory
C++	(adr --)	H	Increment the byte in hub memory
W--	(adr --)	H	Decrement the word in hub memory
W++	(adr --)	H	Increment the word in hub memory
--	(adr --)	H	Decrement the long in hub memory
++	(adr --)	H	Increment the long in hub memory
U@		X	
U!		X	
ALIGN	(adr align -- adr1)	X	Align address upwards to match alignment boundary (i.e. $474A $40 ALIGN .WORD 4780 ok)

MATHS

+	(a b -- c)	C	c = a + b [1234 5678 + . 6912 ok]
-	(a b -- c)	C	c = a - b [6912 5678 - . 1234 ok]
*	(s1 s2 -- s3)	C2	Signed multiply [-1234 5678 * . -7006652 ok]
/	(div1 div2 -- rslt)	X	Signed divide
U/	(u1 u2 -- u3)	H	Unsigned divide
MOD	(a mod -- rem)	X	Extract the remainder after division
UM*/	(um1 um2 udiv -- rsltL rsltH)	X	Multiply um1 by um2 with 64-bit intermediate divided by udiv for a 64-bit result
*/	(um1 um2 udiv -- rslt32)	H	Multiply um1 and um2 to produce a 64-bit intermediate result divided by udiv for 32-bit result
UM*	(u1 u2 -- u1*u2L u1*u2H)	C	unsigned 32bit * 32bit multiply --> 64bit double result
U/MOD	(u1 u2 -- rem quot)	H	Unsigned modulo divide includes remainder [1024 10 U/MOD . SPACE . 102 4 ok]
UM/MOD	(dvndL dvndH dvsr -- rem quot)	C2	Same as UM/MOD64 but constructed with (bytecodes UM/MOD64 DROP)
UM/MOD64	(dvnL dvdH dvsr -- rem qL qH)	C	Full 64-bit by 32-bit divide - used by U/ and U/MOD
UM/MOD32	(d.dvn32 dvsr -- rem qL qH)	C	Same as UM/MOD64 but only uses 32-bits although these have to be left justified
ABS	(a -- b)	C	Absolute value of a - if a is negative then negate it to a positive number
-NEGATE	(a b -- c)	C	Negate a if sign of b is negative (used in signed divide)
?NEGATE	(a flg -- b)	C	Negate a if flg is true
NEGATE	(a -- 0-a)	C	Negate a - that is subtract a from zero
1+	(a -- a+1)	C	Increment a unsigned
1-	(a -- a-1)	C	Decrement a unsigned
2+	(a -- a+2)	C2	Increment a by 2 unsigned (actually a double bytecode instruction 1+ 1+)
2-	(a -- a-2)	C2	Decrement a by 2 unsigned (double bytecode instruction)
4*	(a -- a*4)	C2	Shift a 2 bits left unsigned (double bytecode instruction)
RND	(-- rnd)	X	Generate a 32-bit pseudo-random number enhanced with the system counter
AVG	(val var -- avg)	X	Accumulate the average using 25% of the difference between the current average and val
MIN	(n1 n2 -- n3)	X	Return signed minimum of two items (NOTE! not the same as some langauges)
MAX	(n1 n2 -- n3)	X	Return signed maximum of two items

CONVERSION

B>L	(a b c d -- dcba)	X	Merge four bytes into one long ($12 $34 $56 $78 B>L .LONG 7856.3412 ok)
B>W	(bytel byteh -- word)	X	Merge bytes into a word
W>L	(wordl wordh -- long)	X	Merge words into a long
W>B	(word -- bytel byteh)	X	Split a word into bytes
L>W	(long -- wordl wordh)	X	Split a long into words
>W	(n -- word)	X	Mask n to a 16-bit word (eq. $FFFF AND)
BITS	(n1 bits -- n2)	X	Extract lsb bits from long (-1 9 BITS .LONG 0000.01FF ok)

LOOPING

DO and LOOP use a separate loop stack to hold the parameters and a branch stack to hold the looping address for very fast looping. So loop indices are available outside of the loop as when functions are called from inside the loop. The words associated with DO and FOR are actual instructions which do not need to calculate branch addresses immediately at compile time. These instructions push their current IP onto the branch stack and use this for fast and efficient looping. Take care with unstructured exiting from loops, use LEAVE or UNLOOP.

FOR	(cnt --)	C	Push cnt onto loop stack and save next IP onto branch stack for NEXT
NEXT	(--)	C	Decrement loop IX and exit loop if IX = 0 or else branch using saved FOR branch address
FROM	(from --)		
BY	(by --)		
I	(-- index)	C	Push DO index onto data stack
K	(-- index2)	H	Push second level DO index
J	(-- index3)	H	Push third level DO index
LEAVE	(--)	H	Set the index to limit-1 so that it will LEAVE the loop when it encounters LOOP

CONDITIONAL BRANCH & LOOPING

IF	(flg --)	HI	IF flg is true (non-zero) then execute between IF THEN or IF ELSE During compilation this leaves the address of the next instruction IF merged with $1F.0000
ELSE		HI	IF flg was not true then execute between ELSE THEN During compilation this checks and resolves a preceding IF and sets up for a THEN
THEN		HI	THEN continue on executing normally (terminates an IF) Check and resolve any IFs or ELSEs and set the forward branch offset

  ENDIF

BEGIN		HI	BEGIN a conditional loop - marks the spot for a BEGIN UNTIL or BEGIN WHILE REPEAT During compilation this leaves the address of the next instruction merged with $BE.0000
UNTIL	(flg --)	HI	UNTIL flg is true continue back to matching BEGIN (BEGIN.......UNTIL)
AGAIN		HI	Jump back AGAIN to first instruction after matching BEGIN (BEGIN......AGAIN)
WHILE	(flg --)	HI	WHILE flg is true continue executing code up to REPEAT (BEGIN......WHILE.....REPEAT)
REPEAT		HI	REPEAT the conditional loop by jumping back to after matching BEGIN

I/O PORTS

OUTSET	(mask --)	C	Set the pins as high outputs (also sets DIR bits)
OUTCLR	(mask --)	C	Clear the pins to low outputs (also sets DIR bits)
INPUTS	(mask --)	C	Float the pins to inputs.
SHROUT	(mask dat -- iomask dat/2)	C	Shift out right the lsb of dat over the pins in iomask and return with the shifted data
SHRINP	(iomask dat -- iomask dat/2)	C	Shift in right into msb of dat using iomask to specify the pin.
CLOCK		C	.
OUTPUTS	(mask --)	C	Set the pins to outputs (normally redundant)
(WAITPEQ)	(--)	C	wait for pin state equal to mask stored in COGREG 3
(WAITPNE)	(--)	C	wait for pin state NOT equal to mask stored in COGREG 3
LOW	(pin --)	X	Clear pin low as an output
HIGH	(pin --)	X	Set pin high as an output
FLOAT	(pin --)	X	Float the pin (make it an input)
PINS!	(data pin pins --)	X	Write the bits in data aligned to pin for pins (i.e. %1101 16 4 PINS!)
MASKS	(pin pins -- mask)	X	Build a mask aligned to pin for pins
PINS@	(pin pins -- n)	X	Read from pin for pins and right justify result
LOW?	(pin -- flg)	X	Test if pin is low
HIGH?	(pin -- flg)	X	Test if pin is high
IN	(pinmask -- flg)	X	Test pins using mask
PIN@	(pin -- flg)	X	Test state of pin
OUT	(data pinmask --)	X	Set pins in pinmask to outputs and write data to them

```
PIN!          ( state pin -- )              X    Set pin to state (i.e. ON 6 PIN!)
P!            ( long -- )                   X    Write directly to OUTA
P@            ( -- long )                   X    Read directly from OUTA
```

CALLS AND BRANCHING

```
RESET                                       C    Reset this cog only
0EXIT         ( flg -- )                    C    Exit if flg is zero. This saves a IF … THEN ;
?EXIT         ( flg -- )                    H    Exit if flg is true.
EXIT                                        C    Exit from a called routine and pop the return stack into the IP
CALL          ( adr -- )                    C    Call the adr - used to execute cfa vectors
JUMP          ( adr -- )                    C    Same as CALL but doesn't save the return address
EXECUTE                                     H
XCALLS        ( -- adr )                    H    Return with the address of the XCALLS table which holds the vectors that point to the HL code
```

RETURN STACK

```
!RP                                         C2   Init return stack pointer
>R            ( a -- )                      C    Push a from data stack onto return stack
R>            ( -- a )                      C    Pop a from return stack onto data stack
```

SPI INSTRUCTIONS

These are fast optimized bytecode instructions for reading and writing an SPI bus whose parameters are held in COGREGS - use SPIPINS to set. Most parameters can be reused as in multibyte shifts plus this makes the transfer faster as pushing and popping the data stack slows things down.

```
SPIWRB        ( byte -- byte )              C    Send byte over SPI lines as defined in COGREGs and return with same byte
SPIWR16       ( long -- long1 )             C    Send msb 16-bits (b31..b16) over SPI and return with long rotated left by 16 bits
SPIWR         ( long -- long1 )             C    Send 8 MSBs of long over SPI and return with left rotated long1 ( $12345678 -> $34567812 )
SPIRD         ( long -- long1 )             C    Read SPI data and left rotate into long with long1 as result ( $12345678 -> $345678NN )
SPICE         ( -- )                        C    Release the SPI CE line (automatically enabled on any SPI operation)
SPIPINS       ( &ce:miso:mosi:clk -- )      X    Set pins to be used by SPI - parameter is encoded as byte fields - use & prefix to force decimal bytes
```

I/O MASKS

```
MODPINS       ( pins -- )                   X    Set the pin masks for RUNMODs using (&27.26.25.23 MODPINS to set ce.miso.mosi.clk)
SETPINS       ( pins adr -- )               X    Set pins masks using adr for cog starting address of clk pin

@SCL          ( -- -6 )                     X    COGREG address for SCL (used for fast CLOCK instruction)
@CE           ( -- 3 )
@MISO         ( -- 2 )
@MOSI         ( -- 1 )
@SCK          ( -- 0 )                      X    COGREG address for SCK mask
@CNT          ( -- 4 )                      X    COGREG address for variable CNT used by some RUNMODs
```

COG INSTRUCTIONS

```
LOADMOD       ( src dst cnt -- )            C2   Load cog memory from hub memory - used internally by CODE MODULES
RUNMOD                                      C    Run the currently loaded code module
COGID         ( -- id )                     C2
(COGINIT)                                   C2
COGINIT       ( code pars cog -- ret )      H    Same as COGINIT in PASM - also saves information in cog TASK block (8 bytes/cog)
COG@          ( adr -- long )               C2   Fetch long from cog memory
COG!          ( long adr -- )               C2   Store long to cog memory
COGREG        ( index -- adr )              C2   Return with address of COGREG
REBOOT                                      H    Reboot the current cog
STOP          ( cog -- )                    H    Stop the select cog
STACKS        ( -- stacks )                 H    Return with base address of stacks in cog (refer to source)
```

CODE MODULES

These are small PASM modules that loaded into once the Tachyon cog and executed repeatedly with the separate RUNMOD word.

```
[SDRD]                                      H    SD card block read
```

```
RUNMOD        ( dst char  --  firstPos charcnt )
Read block from SD into memory while scanning for special char
dst is a 32 bit SD-card address 0..4GB, char is the character to scan for while, reading in the block.
 NOTE: ensure MOSI is set as an output high from caller by  1 COGREG@ OUTSET
 This is just the low-level block read once the SD card has been setup, so it just reads a sector into the dst
 There is also a scan character that it will look for and return its first position and how many were found
```

```
[SDWR]                                      H    SD card block write
```

```
RUNMOD        ( src cnt  --  )
Write a block to the SD card - normally 512 bytes
```

```
[PWM32]                                     H    PWM32 runtime (takes over cog)
```

```
RUNMOD        ( table waitcnt -- )
32 channel 8-bit PWM that runs up to 7.6kHz
Can be used as an arbitrary waveform generator too as it reads a long from table (32 channels) every waitcnt sample and
writes to all the outputs that are enabled in the PWM cog. The normal wave table is 256 longs deep.
The table must be aligned to a 256 long boundary
```

```
[PWM32!]                                    H    PWM32 table setup
```

```
RUNMOD        ( duty8 mask table -- )
Write 8-bit duty cycle to channels specified in mask at specified table
```

```
[PLAYER]                                    H    Simple wave player
```

```
RUNMOD        ( ctls blk -- )
Play 16-bit signed wave samples from 1k block. As each half of the 1k buffer is used it is filled again by another cog
which may only need to poll as required. Normally wave samples are played at 44kHz but slower sample rates may also be
used or the pitch adjusted. Use a simple RC filter such as 220R and 0.1uF which provides a low output impedance and a
10k pullup and pulldown will hold the output at 1/2Vdd to prevent clicks (sample($0000) = 1/2Vdd)
The paraameter ctls points to a control block that holds: volume(2),speed(2)
```

```
[WS2812]                                    H    WS2812 RGB LEDs ( array cnt -- )
```

```
RUNMOD        ( array cnt -- )
pin mask is in COGREG4, line RET is done at HL, not here
Will transmit a whole array of bytes each back to back in WS2812 timing format
A zero is transmitted as 350ns high by 800ns low (+/-150ns)
A one is transmitted as 700ns high by 600ns low
```

```
[WS2812] [SDRDF] [SDRD] [SDWR] [SDIO] [SSD!] [PWM32] [PWM32!] [PLOT] [ROL3] [CAP] [WAV] [MCP32] [RCTIME]

[SSD]                                       H    TFT display
[ESPIO]                                     H    Enhanced Serial Peripheral I/O
[SPIO]                                      H    Serial Peripheral I/O
[MCP32]                                     H    SPI for MCP3208 style chips etc
[PLOT]                                      H    Fast plotting
[BCA]                                       H    Byte Code Analyser for debug
[SQRT]                                      H    Fast SQRT ( n -- sqrt(n) )
[CAP]                                       H    Fast I/O Capture for SPLAT logic analyser ( buf lcnt dly -- )
```

ROMS

```
In V3 ROMS are binary images that are saved to upper EEPROM (or elsewhere) that can be loaded into cogs at runtime by name. The F32 ROM is loaded by
default for floating-point extensions (that are being added).
```

```
lsroms
FINDROM ( name -- addr )
LOADCOG ( pars cog name -- )                      Load cog with ROM if found (ex: vgapars 3 " VGA32x15" LOADCOG)
```

TIMING

```
DELTA         ( n -- )                      C2   Add in the delta to cnt and WAITCNT
WAITCNT                                     C2   Wait until CNT reaches the DELTA value - callo repeatedly after first setting DELTA
LAP           ( -- )                        C2   Latch the CNT value and before saving calculate the difference from previous LAP and save
ms            ( n -- )                      X    Delay for n milliseconds
us            ( n -- )                      X    Delay for n microseconds (+10us overhead but values are compensated so 20 us = 20us )
second        ( n -- )                      X    Delay for 1 or more seconds
  seconds                                         - included for readability as in 3 seconds vs 3 second
```

DEFINITIONS

```
(:)                                         H
:             ( <name> -- )                 HI   Create a Forth definition and compile all words into it until a ; is encountered
  pub
pri           ( <name> -- )                 HI   Create a Forth header exactly the same as : except set the private attribute in the header.
                                                   If RECLAIM is executed later on it will find all headers with the private attribute and strip them out.
;                                           HI   Compile an EXIT instruction before finishing off a definition and unsmudge the latest word
UNSMUDGE                                    HI   While a new word is being defined its header is smudged and unknown until it is complete.
```

COMMENTS

```
\                                           HI   Comment the rest of the line and do not echo - \ this is a comment
  ''                                        HI   similar to Spin comment operator
  ---                                       XI   prefered Tachyon comment as it separates sufficiently and does not look like any other operator
  //                                        XI   typical comment operator used in some languages
  \\\                                       XI   note: normally used to disable a line of code
(                                           HI   Comment up to the matching ) and echo -    ( what follows is a stack comment )    ( n1 n2 -- n3 )
{                                           HI   Ignore all text up to the matching }. Used for multiline comments. Nesting to 255 levels
}                                           HI   Outside of a block comment this symbol will simply be ignored
```

CONDITIONAL COMPILE

```
IFDEF <name >                               HI   IF <name> DEFINED then process all source between here and the matching }
IFNDEF <name>                               HI   IF <name> NOT DEFINED then process all source between here and the matching }
```

COMPILE LITERALS

```
Bytes, words, and longs may be compiled directly into code memory usually for building fixed tables.
These cannot be used inside a definition as any preceding literal would have already been compiled as a literal.

C,            ( byte -- )                   HI   Compile the preceding byte into code memory
|             ( byte -- )                   HI   Alias for C, - less clutter when building tables i.e. 34 | 45 | 98 | 20 | etc
||            ( word -- )                   HI   Compile the preceding word into code memory i.e. $1234 || $0FCA || $0082 ||
,             ( long -- )                   HI   Compile a long as used in building tables i.e. $1234.5678 , $DEAD.BEEF ,

TO            ( val <name> -- )             HI   Assign a new value to an existing constant (variable constant) - only works in interactive mode!
BCOMP                                       HI
```

CASE STATEMENTS

```
CASE statements are constructed in a manner similar to C using SWITCH, CASE, and BREAK.

BREAK                                       HI   Stop executing this CASE code and return immediately from routine
CASE          ( val -- )                    HI   Execute the following code up to BREAK if val = SWITCH val
                                                   i.e. ( $0D CASE PRINT" CARRIAGE RETURN" BREAK )
SWITCH        ( val -- )                    H    Store the switch value in a task variables so that is can be referenced by a CASE statement.
SWITCH@       ( -- val )                    H    Retrieve the switch value - useful if we want to perform more complex comparisons
```

```
SWITCH=       ( val -- flg )                  H    Return true if val = SWITCH
SWITCH><      ( val -- flg )                  H    Return true if val <> SWITCH
```

RADIX WORDS

Numbers entered and printed can be represented in any base (radix) by setting the cog variable "base" to that value.
The three most common bases are predefined.
NOTE: It is recommended that numbers other than decimal are always forced with a prefix or alternatively a suffix such as $0FAD or 0FADh etc.

HEX		H	Switch number base to HEX mode - all input and output will default to HEX unless overridden
DECIMAL		H	
BINARY		H	

RADIX OPERATORS

While not defined in the dictionary radix operators force a number to be recognized in a certain base. The operators may be prefixed or suffixed while the prefixed operators have the advantage that the compiler will compile these immediately as a number rather than search the dictionary first as it would with any other number. Tachyon convention is that all hex number are prefixed with $ with the number base set to decimal by default. All numbers 0 to 9 do not need a prefix as that is redundant and besides some of these single digits are predefined as fast constants making them a single bytecode.

$	HEX prefix - number may contain symbols but must end in a valid digit i.e. $Q00FA
#	Decimal prefix - " " i.e. #P26
%	Binary prefix
H	i.e. 00FAh - number must begin with a decimal digit or zero.
D	i.e. 1234d
B	i.e. 01101110b
^<ch>	Return with the control character literal for the next character i.e. ^Z returns $1A
"<ch>"	Return with the ASCII literal for the enclosed character i.e. "Z" returns $5A

CONTROL KEYS

To speed up interactive testing there are certain control keys that can perform operations.

^X	Repeat previous line (re-executes the previous interactive code)
^C	Reboot
^B	Block dump all of hub memory (wait for it)
^D	Dump stacks, registers, and current compilation area
TAB	Tab as normal but handled like a single space
LF	Ignore (as part of a CRLF)
BKSP	Backspace up to the beginning of a word or else the line (preceding words are already compiled)
<ESC>	Cancel the current interactive line
^Z^Z	Cold start - wipe all extensions bar the kernel although everything is still intact in EEPROM
<BREAK>	Will reboot the processor regardless of what code it is running (system in fact detects 100 "framing errors" in a row)

DEBUG

.S		H	Print the contents of the data stack
.STACKS		H	Print the contents of the four stacks
HDUMP		H	Basic kernel hub memory dump (use DUMP from EXTEND)
COGDUMP		H	Only used to directly dump from cog memory
DEBUG		H	Will dump stacks, registers, and current compilation area. Can also be accessed by a single keystroke ^D (control D)

DUMP MEMORY OPERATIONS

Various words are available for general-purpose dumping of memory in hex format. Normally the memory that the DUMP words examine is hub memory but a modifier may be used before a DUMP is executed to use other types of memory. After any dump the default if set back to hub RAM.
Some modifiers are:
EE = EEPROM
COG = COG MEMORY (LONG)
SD = SD card raw
FS = File System (from the start of an open file)
WIZ = WIZnet chip

DUMP	(adr bytes --)	X	Dump as bytes from current dump device including an ASCII code column (revert back to RAM after)
DUMPW	(adr bytes --)	X	Dump as words (same as DUMP)
DUMPL	(adr bytes --)	X	Dump as longs (same as DUMP, formatted as 0000.0000)
DUMPA	(adr bytes --)	X	Dump ASCII printable characters - default width of 64 characters/line (uses . for non-printable)
dwidth	(-- const)	X	Constant value of 64 controls DUMPA width - can be user overridden to suit listing
QD	(adr --)	X	Quick Dump two lines of standard DUMP data
BDUMP	(addr cnt buffer --)	X	Dump listing from a buffer but use source addresses and forced format
bdw	(-- adr)	X	controls word length of dump listing (bytes,words,longs)
dmm	(-- adr)	X	The vectors to methods for device dumps bytes, words, and longs
RAM		X	Change DUMP device to standard HUB RAM (DUMP always defaults here after every DUMP)
EE		X	Change DUMP device to EEPROM (>64k addresses next device etc)

STREAMING I/O

Character based devices such as serial, VGA, LCD etc are treated as streaming I/O where the device code automatically detects and handles special characters. EMIT words will send a single character via the currently selected output device. Conversely KEY is the input from the device.

(EMIT)	(ch --)	C2	The default emit code if uemit is zero
EMIT	(ch --)	H	emit the character via the vector at uemit
CLS	(--)	H	Clear the screen (ANSI)
SPACE	(--)	H	Emit a space
BELL	(--)	H	Emit a bell character
CR	(--)	H	Emit a CR+LF sequence
SPINNER	(--)	H	Emit the next character in a spinner sequence (| / - \ ) using backspace to reposition
READBUF	(bufadr -- ch+$100)	H	Read a character from the buffer and if one is found add $100 to mark it as active, else return 0
KEY	(-- ch)	H	Read a character from the device, a null indicates that no character was available
WKEY	(-- ch)	H	Always WAIT for a KEY so that even a null is a character
(KEY)	(-- ch)	H	Read the console input, this is the default execution vector when ukey = 0 (see task registers)
VER	(-- adr)	H	Address of longs that holds current kernel version build i.e. VER @ .DEC 27150908 = V2.7 150908

.VER		H	Print verbose Tachyon version number i.e. .VER Propeller .:.:--TACHYON--:.:. Forth V27150908.1000
[CON		X	Switch to console but save current output device - use to print console messages without changes
CON]		X	Restore previous output device before the [CON word was executed
UPPER	(char -- upper)	X	Convert the character to an uppercase character if possible else leave unchanged (if <$60)
WAITKEY	(cnt -- key)	X	Wait for a key but timeout if not received and return instead with a null character
EMITS	(char cnt --)	X	Print the char repeatedly for cnt
SPACES	(cnt --)	X	Print spaces repeatedly for cnt
CTYPE	(str cnt --)	X	Print the string for cnt characters, normally in Forth this is simple TYPE but that is used in FTP
GETLINE	(buf -- len)	X	Read a line into the buffer until an end-of-line and return with len
ESC?	(-- flg)	X	Return true if the last console key pressed was an escape? (even if it's still buffered)
TAB		X	Emit a single TAB
<CR>		X	Emit a single CR (no LF)
KEY!	(char --)	X	Force a character to be read as the next KEY
KEY@	(-- char)	X	Read the last console key that has been buffered

BLOCK LOAD

TACHYON		H	Identifies that this Forth source is strictly Tachyon dialect and also to start block load mode.
END			End block load mode and report

CONSTANTS and VARIABLES

:=	(val <name> --)	XI	Create a constant (preferred format reduces clutter around values and names)
TABLE	(<name> --)	XI	Create a table with zero entries (use , | || words to add entries)
bytes	(cnt <name> --)	XI	Create a byte array
words	(cnt <name> --)	X	Indirectly called by WORDS which performs another action if there is no name
longs	(cnt <name> --)	XI	Create a long array (long aligned)
byte	(<csv> --)	XI	Create byte variables from the CSV list (or just a single variable) (BYTE xy,myvar,net)
word	(<csv> --)	XI	Create word variables (word aligned)
long	(<csv> --)	XI	Create long variables (long aligned)
org	(adr --)	X	Set the data origin for DS style data memory allocation
@org	(-- ptr)		Pointer for org
ds	(bytes <name> --)	XI	Create a constant with the current value of the ORG then advance it by bytes for next DS
ds+	(bytes --)	X	Allocate bytes at org

STRINGS

Strings are represented in Tachyon as an address to a null (or 8th bit) terminated string.

STREND	(str -- strend)	C	Return with the address of the end (null) of the string
CMPSTR		C2	
PRINT$	(str --)	H	Print out the null terminated string at str onto the currently selected output device (via uemit)
LEN$	(str -- len)	H	Return with the length of the null terminated string
"	(<str>" -- str)	HI	Process the following characters up to " as a string and leave the address on the stack. Outside of a definition the string buffer will be reused and not available after the line is processed.
."	(<str>” --)	HI	Print the literal string. Example: PRINT" HELLO WORLD" -- actually the compile-time part of it
PRINT$		X	This form is preferred for readability as "dot" words are sometimes harder to pick out
(.")	(--)	HI	this is the helper which does the runtime printing
CALL$	(str --)	X	Find the string in the dictionary if it exists and call it
FIND	(str -- cfa)	X	Find the cfa of this string if it exists in the dictionary else return with zero
>CSTR			
COMPARE$	(str1 str2 -- flg)	X	Compare two strings for equality
$=			
LOCATE$	(ch str -- str)	X	Locate the first ch in the string and return else null
ERASE$	(str --)	X	Fully erase the string - reads max len from header
RIGHT$	(str len -- str)	X	give a copy of the rightmost len chars of str
LEFT$	(str len -- str)	X	Destructive LEFT$ - uses same string
MID$	(str offset len -- str)	X	Extract the substring of str starting at offset len chars long
+CHAR			
APPEND$			
COPY$	(str1 str2 --)	X	Copy (store) str1 to str2
$!			
NULL$	(-- str)	X	Just an empty string
STRING	(str max --)	XI	Immediate word to build a string with a maximum size (use 0 to fit current length)

PRINT NUMBERS

.HEX	(n --)	X	Print the nibble in n as a single hex character
.BYTE	(n --)	X	Print the byte in n as two hex characters
.WORD	(n --)	X	Print the word in n as four hex characters
.LONG	(n --)	X	Print the number in hexadecimal as a long (i.e. 0 @ .LONG 05B8.D800 ok)
.	(n --)	X	Print number unformatted in the current base
@.	(adr --)	X	Fetch long and print value in current base
>DIGIT	(char -- val true | false)	X	Convert ASCII value as a digit to a numeric value (only interested in bases up to 16 at present)
NUMBER	(str -- val digits | false)	X	Convert a string to a number if possible
@PAD	(-- adr)	X	Pointer to current position in number pad
HOLD	(ch --)	X	Prepend the character to the number string buffer
>CHAR	(val -- ch)	X	Convert a binary value to a character that represents that digit (0-9,A-Z,a-z)
#>	(a -- str)	X	Stop converting the number and discard what's left and return with a ptr to the string
<#		X	Start converting a number to a string by resetting the number buffer
#	(a -- b)	X	Extract another digit from the a leaving b and prepend the digit to the number string buffer
#S	(a n --)	X	Extract n digits using # word
<D>	(--)	X	Signal that the current number is to be printed should be processed as a double number
U.	(u1 --)	X	Print unsigned number
.DEC	(u1 --)	X	Print unsigned number as decimal with at least 4 digits (for formatting)
NUM>STR	(num -- str)	X	Convert a number to a string and buffer it in NUM$ where it can be manipulated etc
NUM$	(-- adr)	X	NUM>STR holding buffer up to 15 digits long (+terminator) - can be referenced again if needed..
U.N	(number digits --)	X	Print an unsigned number in the current base with at least this many digits
.ADDR		X	Print as 8 digit hex address formatted as (0000.0000:)
.INDEX		X	Print the current DO index on a new line as (0000:)

```
PRINTDEC      ( number digits -- )             X    Print a formatted decimal number with at least this many digits
.DECX         ( number -- )                    X    Print a formatted decimal number with 4 digits
.DP           ( dblnum decimals --- )          X    Print the double number with decimal places (scaled)
.NUM               ( number format -- )        X    Print number according to format spec (see table)
   PRINTNUM
```

```
Format the number as to it's base, the number of digits, whether it's signed, and should it have separators
This is a very versatile print number routine and can be incorporated into other print words
A single word is passed with options:
b07..00 = base
b12..08 = digits
b13     = use spaces in place of leading zeros
b14     = force separators (decimals have a , every 3 places while others have a _ every 4 places)
b15     = force sign (always + or - depending upon the number)
Usage:
    $C00A will force separators, sign, no leading zeros (0 digits), and a base of 10
i.e.    0 @ $C00A .NUM +80,000,000 ok
    $4810 will force separators, no sign, and 8 digits with a base of 16
i.e.    0 @ $4810 .NUM 04C4_B400 ok
```

TASK VARIABLES

Each cog may have its own set of variables that are offset from the address in COGREG 7.
This is so that any cog running Tachyon may have different I/O devices selected etc.
Only a small number of these variables are named in the dictionary but they can be referenced from these with an offset by referring to the source.

```
REG           ( index -- adr )                 C    Find the address of the register for this cog
rx                                             H    Pointer to the rx buffer with the 2 words before the buffer as rxrd and rxwr index
flags                                          H    Bit flags used by the kernel
          ❏ echo                                = 1      ' managed by pub ECHO  \ ON ECHO \ OFF ECHO for console echo
          ❏ linenums                            = 2      ' prepend line number to each new line
          ❏ ipmode                              = 4      ' interpret this number in IP format where a "." separates bytes
          ❏ leadspaces                          = 4
          ❏ prset                               = 8      ' private headers set as default
          ❏ striplf                             = $10    ' strip linefeeds from output if set ( not used - LEMIT replaces this !!!)
          ❏ sign                                = $20
          ❏ comp                                = $40    ' force compilation of the current word - resets each time
          ❏ defining                            = $80    ' set flag so we know we are inside a definition now

base          ( -- adr )                       H    Byte variable specifying the current base + backup byte used during overrides
digits        ( -- adr )                       H    Byte variable with count of digits from last number parsed
delim         ( -- adr )                       H    Word delimiter (normally space) plus backup byte with delimiter detected (SP,TAB,CR etc)
word          ( -- adr )                       H    Pointer to word buffer where a parsed word is stored
switch        ( -- adr )                       H    SWITCH value is stored as a long here (single level only)
autorun       ( -- adr )                       H    Pointer to cfa of user autostart routine normally used by EXTEND which implements a new user vect
keypoll       ( -- adr )                       H    User app may set this to the cfa of a routine that gets polled while KEY is idling.
tasks         ( -- adr )                       H    Holds task list of 8 bytes for each 8 cogs (IP[2],RUN[1] implemented in EXTEND
unum          ( -- adr )                       H    User number processor vector. 0 defaults to kernel method.
uemit         ( -- adr )                       H    Vector points to cfa of current EMIT routine (0=console=(EMIT))
ukey          ( -- adr )                       H    Vector points to cfa of current KEY routine (0=console=(KEY))
names         ( -- adr )                       H    Points to the start of the latest name field in the dictionary (builds down)
here          ( -- adr )                       H    Points to the end of the code space but normally referenced by HERE (here W@)
codes         ( -- adr )                       H    Temporary code space pointer while a line is compiled but not yet committed (interactive)
baudcnt       ( -- adr )                       H    EXTEND has SERIN and SEROUT routines which store their baudrate CNT value here for each cog
prompt        ( -- adr )                       H    User vector may point to code to change the prompt (normally blank)
ufind         ( -- adr )                       H
create        ( -- adr )                       H
Lines         ( -- adr )                       H    holds line count during block load
errors        ( -- adr )                       H    holds count of errors detected during block load of source via TACHYON word
lastkey       ( -- adr )                       H    The last key that was pressed from the serial console is stored here, useful for lookaheads.
```

Registers by index

```
' Minimum registers required for a new task - other registers after the ' ---- are not needed other than by the console
0         temp                res 12            ' general purpose
12        cntr                res 4             ' hold CNT or temp
' @16
16        uemit               res 2             ' emit vector - 0 = default
18        ukey                res 2             ' key vector
20        keypoll             res 2             ' poll user routines - low priority background task
22        base                res 2             ' current number base + backup location during overrides
24        baudcnt             res 4             ' SERIN SEROUT baud cnt value where baud = clkfreq/baudcnt - each cog can have it's own
28        uswitch             res 4             ' target parameter used in CASE structures
32        flags               res 2             ' echo,linenums,ipmode,leadspaces,prset,striplf,sign,comp,defining
34        keycol              res 1             ' maintains column position of key input
35        wordcnt             res 1             ' length of current word (which is still null terminated)
36        wordbuf             res wordsz        ' words from the input stream are assembled here
' numpad may continue to build backwards into wordbuf for special cases such as long binary numnbers
75        numpad              res numpadsz      ' Number print format routines assemble digit characters here - builds from end - 18,446,744,073,709,551,615
101       padwr               res 1             ' write index (builds characters down from lsb to msb in MODULO style)

'
' ------ console only registers - not required for other tasks - so no need to allocate memory beyond here
'



102       unum                res 2             ' User number processing routine - executed if number failed and UNUM <> 0
104       anumber             res 4             ' Assembled number from input
108       bnumber             res 4
112       digits              res 1             ' number of digits in current number that has just been processed
113       dpl                 res 1             ' Position of the decimal point if encountered (else zero)
```

```
' WORD aligned registers

114       ufind               res 2             ' runs extended dictionary search if set after failing precompiled dictionary search
116       createvec           res 2             ' If set will execute user create routines rather than the kernel's

118       rxptr               res 2             ' Pointer to the terminal receive buffer - read & write index precedes
120       rxsz                res 2             ' normally set to 256 bytes but increased during block load
122       corenames           res 2             ' points to core kernel names for optimizing search sequence
124       oldnames            res 2             ' backup of names used at start of TACHYON load
126       names               res 2             ' start of dictionary (builds down)
128       prevname            res 2             ' temp location used by CREATE
130       fromhere            res 2             ' Used by TACHYON word to backup current “here” to determine code size at end of load
132       here                res 2             ' pointer to compilation area (overwrites VM image)
134       codes               res 2             ' current code compilation pointer (updates "here" or is reset by it)
136       cold                res 2             ' pattern to detect if this is a cold or warm start ($A55A )
138       autovec             res 2             ' user autostart address if non-zero - called from within terminal
140       errors              res 2
142       linenum             res 2

' Unaligned registers

144       delim               res 2             ' the delimiter used in text input and a save location

146       prompt              res 2             ' pointer to code to execute when Forth prompts for a new line
148       accept              res 2             ' pointer to code to execute when Forth accepts a line to interpret (0=ok)


150       prevch              res 2             ' used to detect LF only sequences vs CRLF to perform auto CR
152       lastkey             res 1             ' written to directly from serialrx
153       keychar             res 1             ' override for key character

154       spincnt             res 1             ' Used by spinner to rotate busy symbol

155       prefix              res 1             ' NUMBER input prefix
156       suffix              res 1             ' NUMBER input suffix
157                           res 3

160       tasks                res tasksz*8     ' (must be long aligned)

224       endreg               res 0
```

DICTIONARY

The Tachyon names dictionary is a completely separate area from the code dictionary and is where all the names, name and code attributes, and code "pointer" is stored. Unlike traditional Forths the names are not stored inline with the code and this allows for more flexibility with memory and dictionary. For instance names may be removed without touching or impacting code and also aliases may be added that are simply clones of the original header so they do not add any overhead. Also names may be declared as private allowing them to be removed later on and the memory reclaimed. Because code no longer has inline headers and is always bytecode it also means that code can "fall through" into the next code forward definition thus simplifying code.

Here is an example of a code header which has the fields: count(1), name(V), ATR(1), pointer(2)

```
: HELLO PRINT" HELLO WORLD"  CR ;  ok
@NAMES 10 DUMP
0000_559D:   05 48 45 4C  4C 4F 82 BD   53
```

Dictionary ATR fields always have the msb set thus terminating the name if it is referenced as a string since both nulls and >$7F characters are valid string terminators in Tachyon. The pointer is more commonly not a pointer as such but the bytecodes to be compiled, either one or two bytes. The name count is used both for quick linking into the next word (nextadr = adr+cnt+4) and also to speed up searching as string compare is skipped if counts do not match. Name dictionaries build down from a high address toward the code dictionary which builds up (since it needs to execute in this manner) from a low address towards the name dictionary. So free space is the difference between the latest entry in the names dictionary @NAMES and the current HERE which points to the next free code location.

```
CREATE$                                    X    Create a new header in the dictionary
ALLOT         ( n -- )                     H    Allot n bytes of code memory - advances "here"
ALLOCATED                                  H
HERE          ( -- adr )                   H
>VEC                                       H
>PFA                                       H
[NFA']                                     H
[']                                        H
ERROR                                      H
NOTFOUND
NFA'          ( <word> -- nfa )            H    Return with the name field address of the following word
'             ( <word> -- cfa )            H    Return with the code address of the following word. If cfa<$100 = bytecode address in cog memory.
SCRUB                                      H
GETWORD       ( <word> -- str )            H    Wait for a terminated word to be entered and return with the ptr to the wordbuf
SEARCH        ( cstr -- nfaptr )           H    Search the dictionaries for cstr which points to the word string constructed as count+string+null
FINDSTR       ( cstr dict -- nfaptr | false )  H    Try to find the counted string in the dictionary using CMPSTR  (ignore smudged entries)
NFA>CFA                                    H
LITERAL                                    H
(CREATE)                                   H
CREATEWORD                                 H
CREATE                                     H
[COMPILE]                                  H
GRAB                                       H
TASK                                       H
IDLE                                       H    Start-up used by idle cogs which checks for a run address while pausing to save power
LOOKUP                                     H
+CALL                                      H
FREE                                       H
TERMINAL                                   H
CONSOLE                                    H
```

```
DISCARD                                     H

@NAMES        ( -- adr )                    X     Return with point to start of latest dictionary header (builds down)

XCALL  0E h.i...x. {HELP
XCALL  0E h.i...x. HELP:
YCALL  08 h.i...x. ALIAS
YCALL  07 h.....x. PUBLIC
YCALL  06 h.....x. PRIVATE
YCALL  05 h.i...x. IMMEDIATE
YCALL  04 h...p.x. @HATR
YCALL  03 h.....x. @NAMES
```

BUFFERS

```
BUFFERS                                     H     2k bytes available for up to 4 open files or general use
@VGA          ( -- adr )                    H     Pointer to VGA buffer used by kernel's VGA driver
```

SPECIAL DEVICES

```
RGBLEDS       ( buffer cnt pin – )          X     Output buffer for cnt to pin to drive WS2812 RBG LED string
DHT           ( pin -- rh temp )            X     Read DHT22 type humidity/temperature sensor on this pin and use last reading if too soon
PING          ( trig echo -- us )           X     Trigger PING sensor and listen on echo pin and return with microseconds value
DISTANCE      ( trig echo -- distance.mm )  X     PING sensor and return with reading in millimetres
```

RTC

Many I2C RTC chips are very similar in layout and at present there are two types I mainly use but these are mostly compatible with many other types as the registers are in the same place it's just that sometimes some chips use the unused bits for various things. But you can select the main ones I use as the MCP79410 series and the DS3231 which is temperature compensated and includes temperature readings.

```
MCP79410                                    X     Select MCP79410 compatible I2C RTC
DS3231                                      X     Select DS3231 compatible I2C RTC (DS1307 etc)
@rtc          ( -- id )                     X     I2C RTC address 8-bit constant
Rtcbuf        ( -- adr )                    X     pointer to rtc buffer
TIME!         ( time -- )                   X     Store time as decimal hhmmss
TIME@         ( -- time )                   X     Fetch time as decimal hhmmss
DATE!         ( date -- )                   X     Store date as international format decimal yymmdd
DATE@         ( -- date )                   X     Fetch data as yymmdd
DAY!          ( day -- )                    X     Store day of week as day 1 to 7    (MON to SUN)
DAY@          ( -- day )                    X     Fetch day of week
SUN           ( -- 7 )                      X     Day constant - use like this - SUN DAY!
SAT           ( -- 6 )                      X     Day constant
FRI           ( -- 5 )                      X     Day constant
THU           ( -- 4 )                      X     Day constant
WED           ( -- 3 )                      X     Day constant
TUE           ( -- 2 )                      X     Day constant
MON           ( -- 1 )                      X     Day constant
.TIME                                       X     Print the current time as HH:MM:SS
.DATE                                       X     Print the current date as YY/MM/DD
.DAY                                        X     Print day as 3 characters (Mon..Sun)
.DT                                         X     Print date and time
.ASMONTH      ( month -- )                  X     Print month index 1-12 as 3 characters (Jan..Dec)
'C                                          X     Read temperature from DS3231 and display as celsius with two decimal places
'F                                          X     Read temperature from DS3231 and display as Fahrenheit with two decimal places
.DTF
DEC>BCD       ( dec -- bcd )                X     Convert decimal 0..99 to BCD
BCD>DEC       ( bcd -- dec )                X     Convert BCD to decimal 0..99
WRRTC                                       X     Write rtcbuf to RTC
RDRTC                                       X     Read RTC into rtcbuf
RTC@          ( adr -- byte )               X     Read a byte from the RTC register
RTC!          ( byte adr -- )               X     Write a byte into the RTC register
```

TIMERS

Tachyon maintains a background timer cog which counts every millisecond and scans a linked list of user counters that may be setup to simply countup or countdown to zero and optionally execute an ALARM condition. Also system runtime is maintained so this can be quite useful plus soft RTC functions are available too.

```
TIMER         ( <name> -- )                 XI    Create a new timer structure (10 bytes)
TIMEOUT?      ( timer -- flg )              X     Check if this timer (using TIMER name) has timed out. (also links this timer into the list)
ALARM         ( cfa timer -- )              X     Set the alarm condition to be executed when this timer has timed out
COUNTUP       ( timer -- )                  X     Set this timer as a simple up counter every ms
TIMEOUT       ( ms timer -- )               X     Set this timer to timeout in milliseconds - stops at zero and executes ALARM if set
WATCHDOG      ( ms -- )                     X     (Re)Trigger watchdog and timeout in milliseconds to reboot
```

POLLING

Somewhat related to timers but completely different is the background polling which goes on when a cog is waiting for KEY input. In the main console task this allows a user routine to add POLLS that check for low priority tasks that are more able to be handled by the main console cog such as detecting for SD card inserted etc. The user may add up to 8 polls and this needs to be done in the user init routine as all polls are cleared on boot.

```
+POLL         ( cfa -- )                    X     Add the code routine as a background POLL
```

I2C BUS

```
I2CPINS       ( sda scl -- )                X     Setup the pins that will be used for the I2C bus
EEPROM                                      X     Setup P29 and P28 as the I2C bus pins
I2CSTART                                    X     Generate an I2C START condition
I2CSTOP                                     X     Generate an I2C STOP condition
?I2CSTART                                   X     Generate an I2C STOP condition if the bus is free (from another cog)
I2C!          ( byte -- )                   X     Write a byte to the I2C bus
I2C@          ( ack -- byte )               X     Fetch a byte from the I2C bus and write the ack state (as is so that 0 = ack)
I2C!?         ( byte -- flg )               X     Write a byte to the I2C bus and return with the ack state
```

```
ackI2C@       ( -- byte )                   X    Same as I2C@ except the ack state is set active already (= 0 I2C@)
FI2C@                                       X    Fast I2C@
i2cflg
SDA
SCL

EEPROM
EC@           ( adr -- byte )               X    Read a byte from EEPROM at adr    (spans multiple 64k devices)
EC!           ( byte adr -- )               X    Write a byte to the EEPROM at adr   (spans multiple 64k devices)
EW@           ( adr -- word )               X    Read a word from EEPROM (non-aligned)
EW!           ( word -- adr )               X    Write a word to EEPROM (non-aligned)
E@            ( adr -- long )               X    Read a long from EEPROM (non-aligned)
E!            ( long -- adr )               X    Write a long to EEPROM (non-aligned)
ESAVE         ( ram eeprom cnt -- )         X    Save a block of RAM to EEPROM using page write. Will backup 32K to EEPROM in 4.963 seconds )
ESAVEB        ( ram eeprom cnt -- )         X    Save a block of RAM to EEPROM using byte by byte method, slower and safer for non-page alignments
ELOAD         ( eeprom ram cnt -- )         X    Load a block of EEPROM to RAM. Will load 32K from EEPROM in  4.325sec
BACKUP                                      X    Backup all of 32k of hub RAM to $0000 of first EEPROM
?BACKUP                                     X    Only backup if there were no errors in the TACHYON block load
ECOPY         ( eesrc eedst cnt -- )        X    Copy cnt bytes from eesrc to eedst
EVERIFY       ( ram cnt -- )                X    Compare cnt bytes of EEPROM with RAM starting at address ram
EFILL         ( src cnt ch -- )             X    Fill EEPROM from src address for cnt times with byte ch
EE                                          X    Select EEPROM for memory DUMP using various DUMP methods (i.e. 0 $100 EE DUMP)
EERD          ( -- ack )                    X    Switch EEPROM to read mode, check ack
@EE           ( adr -- ack )                X    Select the appropriate device and issue an address, check ack


COLD                                        X    Reset to a kernel only system in RAM without extensions although EEPROM is not affected
AUTORUN <name> XI                           Set system to autorun name if found else clear autorun if invalid. Executed via EXTEND.boot
                                                 If the name does not exist in the dictionary at boot-time it will no longer be valid (as in FORGET <name>)
EXTEND.boot                                 X    First code that is executed outside the kernel, resets polls and timers and checks AUTORUN

RECLAIM                                     X    Scan the dictionary for any private words and removed their headers and reclaim dictionary memory by compacting
CONBAUD       ( baud -- )                   X    Set the startup baudrate of the console into EEPROM, needs restart to activate. Recommend 300 to 2,000,000
.LAP                                        X    Print results of LAP <tests> LAP

XCALL  D8 h.i...x. KEEP
YCALL  D7 h.....x. (FORGET)
FORGET        ( <name> -- )                 XI   Forget all names in the dictionary from this name onwards - from most recent
YCALL  D6 h...p.x. (RECLAIM)
XCALL  D6 h.i...x. STRIP
YCALL  D5 h...p.x. (STRIP)

WORDS                                       XI   NOTE: special case performs one of two functions
 WORDS        ( n <name> -- )                    Allocate n words of variable memory accessed by name
 WORDS                                          List words in dictionary in detail (this form has no trailing spaces or characters)
WWORDS                                      X    List words in dictionary in wide 4 column format without details
QWORDS                                      X    List words in dictionary in quick compact format at around 80 column width (plus overflow)
QW                                          X    List the latest 128 words in quick compact format


ANSI
YCALL  BA h.....x. MARGINS
BOLD                                        X    Enable ANSI bold type (if supported)
REVERSE                                     X    Enable ANSI reverse type
ATR
PLAIN                                       X    Reset all type to plain
XCALL  B8 h.....x. CURSOR
YCALL  B7 h.....x. ERLINE
XCALL  B7 h.....x. ERSCN
CLS                                         X    Hybrid $0C EMIT plus ANSI HOME + ERASE SCREEN
HOME                                        X    ANSI home
XY            ( x y -- )                    X    ANSI XY cursor positioning (1 1 XY = home)
PAPER         ( color -- )                  X    Set paper color (background 0..7)
PEN           ( color -- )                  X    Set pen color (foreground 0..7)
white         ( -- 7 )                      X    Constant for ANSI pen or paper color (white PEN)
cyan          ( -- 6 )
magenta       ( -- 5 )
blue          ( -- 4 )
yellow        ( -- 3 )
green         ( -- 2 )
red           ( -- 1 )
black         ( -- 0 )


EASYFILE FAT32
The FAT32 file layer is built on top of basic buffered sector layer and the virtual memory layer on top of that. Since the virtual memory is limited to 4GB
using a 32-bit address a further step is taken to allow a file to be addressed as virtual memory of up to 4GB. There are sector and directory buffers for up
to four files which are opened in the sense that the virtual memory address to the start of the file is located. File sectors are assumed to be contiguous
without fragmentation and this is normally the case as I have never found a fragmented SD card before. Not having to follow clusters simplifies and speeds
the virtual memory layer.

FCLOSE                                      F    Close the current file
FOPEN         ( <file> -- )                 FI   Interactively open a file
FOPEN$        ( file$ -- sector )           F    Open the file specified in the string file$ and return with starting sector or zero if not
FOPEN#        ( diradr -- sector )          F    Open the file pointed to by the virtual directory entry address
RW                                          F    Mark the currently open file read/write
APPEND                                      F    Scan the file for EOF (null) and set write pointer to this location

FPUTB         ( byte -- )                   F    Write a byte into the logical end of the file and update the write pointer
FPUT          ( ch -- )                     F    Like FPUTB but ignore null characters
FGET          ( -- ch )                     F    Read the next character from the file (used by FINPUT)
FILE>                                       F    Set current file as an input device (instead of from console etc)
  FINPUT
```

```
>FILE                                   F    Set current file as an output device (instead of to console etc)
  FOUTPUT

cat           <file>                    FI   Command line "cat" command to list the contents of a file
(cat)
FPRINT$
FRUN
FLOAD         <file>                    FI   COmmand line File load - loads source file which is executed and/or compiled
DELETE        <file>                    FI   Command line file delete
RENAME        <from> <to>               FI   Command line file rename
RENAME$       ( from$ to$ -- )          F    Rename the file using two string parameters

fkey
FREM
@FWRITE
@FREAD

YCALL  5B h.....x. PFA>NFA

CNT@ ( -- cntr )                       Fetch the current contents of the system counter
WAITPEQ
WAITPNE
COGREG@       ( adr -- long )           X    Fetch the contents of the indexed COGREG which is offset from REG0
COGREG!       ( long adr -- )           X
```

UNSORTED

```
.SW                                     H
uboot                                   H
boot                                    H
EVAL$                                   H
(EVAL)                                  H
evio                                    H
evp                                     H

.AUTORUN                                H
.BOOTS                                  H
.STATS                                  H
END                                     H
%L                                      H
.BUILD                                  H
build.date                              H
build.time                              H
build                                   H
lsclk                                   H    List system clock configuration
.diff                                   H
resets                                  H
FIXBIN                                  H
DHTBYTE
DHTBIT
htck
htsav
rh
htref
httime



TIMERTASK
TIMERJOB
timerjob
timerstk
CountDown

time
runtime
wdt

+TIMER
ttint
tid
timers
```