{{

.:.:-- TACHYON --:.:.

A very fast and very small Forth byte code interpreter for the Propeller chip.
2012 Peter Jakacki

TABLE OF CONTENTS

STARTUP SCREEN

```
  Propeller .:.:--TACHYON--:.:. Forth V24140723.0130

NAMES:  $6143...74CF for 5004 (2542 bytes added)
CODE:   $0000...2961 for 5589 (4192 bytes added)
CALLS:  0583 vectors free
RAM:     14306 bytes free

MODULES LOADED:
1901: EXTEND.fth              Primary extensions to TACHYON kernel - 140723-1400
BOOT: EXTEND.boot

-----------------------------------------------------------------
```

FEATURES

V2

- Stacked backward branch references - fast and simple looping
- Fast load features
- Large receive buffer
- Top 4 stack parameters are accessed as fixed registers - overflow into hub RAM
- Small memory footprint
- Low level words and run-time bytecode interpreter in PASM
- Byte codes are read from hub RAM and directly address first 256 longs of PASM code in cog
- User PASM mode via stacks
- Interpreted bytecode definitions are referenced as:
  2 bytes - vectored CALL opcode + byte index into 512 entry table

- All literals and strings are byte aligned
- Fast access Forth registers avoids deep stacks and juggling
- Fast I/O bit-bashing support for clocked and asynchronous data (2.8MHz clock speed)
  Flexible SPI or I2C PASM code support words in kernel
  Construct fast serial drivers with minimal code
- VM Kernel compiled in standard manner via Spin tools so other Spin objects can be combined
- Four stacks in COG RAM: Data, Return, Loop, and Loop Branch
  Access loop indices outside of definitions allows efficient factoring
  Avoids manipulation and corruption of return stack
  Static stack arrays for direct addressing of stack items
  Intrinsically safe data stack overflow and underflow - optional error reporting
- 400ns minimum instruction cycle time (single bytecode, hub access average)
  Empty loops can execute in 550ns to 875ns (absolute worst case)
  Two to one stack operations (+ * AND etc) inc opcode fetch take 900ns to 1.087us (absolute worse case) *
- Flexible number input using common symbols for prefixes and suffixes as well as allowing intermixed symbols

V2.4
Adds KEY! (KEY)
Modifies console KEY now returns with null if there was no key - KEY?

MEMORY MAP

(Link to color coded expanded map with file and network)
NORMAL MEMORY MAP WITH KERNEL + FULL EXTEND.fth

.STATS
NAMES: $5D21...741A for 5881 (3448 bytes added)
CODE: $0000...333E for 7237 (6846 bytes added)
CALLS: 0484 vectors free
RAM: 10723 bytes free

.MAP
REGISTERS x16
0024: .. .. 02 .. .. 01 02 03 02 ..
VECTORS x64
0124: 0E 11 14 10 13 17 15 0F 14 10 17 11 14 14 16 17
0524: 11 .. ..
KERNEL CODE x256
0928: 64 5F 87 89 60 58 61 6F 70 69 6F 60 65 71 63 3F
CODE x256

1880: 66 58 43 45 5D 77 69 62 69 65 63 66 6E 5F 58 64
2880: 6B 86 7F 6A 53 6F 51 71 65 64 57 .. ..
SPARE @333E: for 10,723
DICTIONARY x256
5D21: 53 68 6F 69 63 66 60 60 61 5F 5C 55 54 47 41 44
6D21: 4C 53 5A 51 57 6F 5A 0F 70 6E 7B 81 6D 90 1E ..
SPARE @7423: for 221
BUFFERS x64
7500: 1C 1D 1D 1C 17 1E 1A 1C 1B 1F 1E 21 1D 1E 21 24
7900: 1F 19 1E 1B 17 27 28 24 20 0C 01 .. ..
RX BUFFER x64
7D00: .. 20 23 1D 1E 04 1E ..
HUB STACK x16
7F60: .. 01 .. ..

SPINNERET MEMORY MAP (WHEN FULLY LOADED WITH EXTEND,HEADERS,SDFILE,W5200,NETWORK,APP)
2014/07/16 15:23:01
NAMES: $5626...7442 for 7708 (-586 bytes added)
CODE: $0000...48CD for 10929 (2433 bytes added)
CALLS: 0132 vectors free
RAM: 3417 bytes free

MODULES LOADED:
3F4C: EASYNET.fth WIZNET NETWORK SERVERS 140615.2300
38EA: W5100.fth WIZNET W5100 driver 140615.0000
2FC7: EASYFILE.fth FAT32 Virtual Memory Access File System Layer V1.1 140626-2100
2AF0: SDCARD.fth SD CARD Toolkit - 140626.1400
2A24: EPRINT.fth Stores PRINT strings in EEPROM 140626.0000
28A1: SPINNERET.fth Spinneret + W5100 HARDWARE DEFINITIONS 131204.1200
1881: EXTEND.fth Primary extensions to TACHYON kernel - 140703-14OO
BOOT: EXTEND.boot
POLL: ?EASYNET
BOOT: EASYNET

.MAP
REGISTERS x16
0024: .. 01 .. .. 02 .. .. 03 02 .. ..
VECTORS x64
0124: 10 0F 16 0F 17 13 10 17 13 14 12 13 14 12 15 14
0524: 16 18 18 17 15 16 18 19 1A 16 18 18 .. ..
KERNEL CODE x256
0928: 64 5D 89 85 63 61 5D 6F 6E 69 72 61 60 71 74 40
CODE x256
1881: 66 55 55 6E 76 76 5E 5B 65 6A 68 83 65 61 60 6E
2881: 54 6B 55 2D 7D 7B 86 38 4F 2E 6A 3B 6D 61 69 7C
3881: 6C 51 63 79 73 72 83 5B 7F 59 84 6C 83 7D 74 87
4881: 83 41 .. .. 30 66 63
SPARE @4A04: for 3,070
DICTIONARY x256
5602: 63 66 5F 5E 5A 58 55 55 5A 54 6F 67 65 66 63 5E
6602: 5F 5D 58 54 4B 42 43 4A 4D 5A 52 54 6B 5D 21 0B
SPARE @744B: for 181
BUFFERS x64
7500: 06 05 .. ..
7900: .. ..
RX BUFFER x64
7D00: 01 0F 0E 0E 0D 0E 0F 0E
HUB STACK x16
7F60: .. 01 01 .. ..

---------- WHAT'S NEW ----------

CHANGELOG
}}
DAT
version **long** **24_140928,1630**
{{
TO DO: Convert decimal point numbers to floating point IEEE-754
Modify kernel so that each kernel can do serial receive - use BUFFERs dynamically

If serial lines always have a line delay we can implement a minimum buffer for a line and use a bytecode receive routine instead or a RUNMOD
A null can always be interpreted as a break as nulls would not normally be transmitted serially
For faster load this could be buffered directly into an SD buffer (3ms) and saved sector by sector then compiled from SD

V2.4
140724 Modified +FIB to become BOUNDS instead

140722 Automatically convert case of failed search word to upper case and try again
140721 Fixed COMMENT words so that the terminator is passed through (no more empty comments needing two CRs)
Reshuffled opcodes between first and second page
Removing KEY? in favour of a standard KEY operation which returns with a false on no key or a key with msb set
If a word fails to be found in the dictionary and it starts with a lower-case character then convert it to uppercase and try again

140717 Adding in MYOP and removing CMPSTR so that it only loads as a RUNMOD during compling blocks of source, otherwise runs in HL bytecode.
Working towards integrating runtime OBEX
V2.3
140717 Renamed CROP to MYOP to allow for user programmable opcode - not ever used in kernel or EXTEND
Optimising STREND and CMPSTR
Decided this should be V2.4 as CMPSTR is to removed from cog kernel and only loaded as a RUNMOD during TACHYON .... END loads

Refer to V2.3 for older changelogs

For the latest links:
Introduction to TACHYON Forth (including other links at the bottom of the page)

}}

{ *** SOURCE CODE *** }

```
DAT

baudrate        long    baud                    ' The baudrate is now stored as a variable

CON
_clkmode        = xtal1 + pll8x
_xinfreq  = 10_000_000                          ' <--- AUTOMATIC 5 or 10MHz operation change at boot
sysfreq         = 80_000_000
baud            = 230400                        ' <-- user change - tested from 300 baud to 3M baud

rxsize          = 512                           ' Serial rx buffer - reuses serial receive cog code image

extstk          = $7F60                         ' locate external data stack at end (overflows but it's ROM)


' Standard Port assignments
scl             = 28
sda             = 29
txd             = 30
rxd             = 31

' Stack sizes
datsz           = 4                     ' expands into hub RAM
retsz           = 22
loopsz          = 8
branchsz        = 6
```

```
CON

numpadsz        = 43                    ' We really only need a large buffer for when long binary numbers with separators are used
wordsz          = 39                    ' any word up to 37 characters (1 count, 1 terminator)
tasksz          = 8                     ' 8 bytes/task RUN[2] FLAG[1]

' flags
echo            = 1
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


DAT

' Allocate storage memory for buffers and other variables
```

{ TASK REGISTERS }

```
' Task registers can be switched easily by setting "regptr" ( 7 COGREG! )
' New tasks may only need room for the first 32 or 72 bytes
```

```
registers               long 0[64]              'Variables used by kernel + general-purpose

                        org  0
' register offsets within "registers". Access as    REG,delim   ...  REG,base ... etc
'
' Minimum registers required for a new task - other registers after the ' ---- are not needed other than by the console
temp                    res 12          ' general purpose
cntr                    res 4           ' hold CNT or temp
' @16
uemit                   res 2
ukey                    res 2
keypoll                 res 2           ' poll user routines - low priority background task
unum                    res 2           ' User number processing routine - executed if number failed and UNUM <> 0
baudcnt                 res 4           ' baud cnt value where baud = clkfreq/baudcnt
uswitch                 res 4           ' target parameter used in CASE structures
' @32
padwr                   res 1                   ' write index (builds characters down from lsb to msb in MODULO style)
numpad                  res numpadsz            ' Number print format routines assemble digit characters here
' @72
wordcnt                 res 1                   ' length of current word (which is still null terminated)
wordbuf                 res wordsz  ' words from the input stream are assembled here
' @112
tasks                   res tasksz*8

anumber                 res 4           ' Assembled number from input
bnumber                 res 4
digits                  res 1           ' number of digits in current number that has just been processed
dpl                     res 1           ' Position of the decimal point if encountered (else zero)
' WORD aligned registers

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

rxptr                   res 2           ' Pointer to the terminal receive buffer - read & write index precedes
rxsz                    res 2           ' normally set to 512 bytes
corenames               res 2
                        res 2           ' backup of names used at start of TACHYON load
unames                  res 2           ' user dictionry method, not implemented yet - needed? ufind
names                   res 2           ' start of dictionary (builds down)
prevname                res 2           ' temp location used by CREATE
                        res 2
here                    res 2           ' pointer to compilation area (overwrites VM image)
codes                   res 2           ' current code compilation pointer (updates "here" or is reset by it)
cold                    res 2           ' pattern to detect if this is a cold or warm start (A55A )
autovec                 res 2           ' user autostart address if non-zero - called from within terminal
errors                  res 2
linenum                 res 2

' Unaligned registers

delim                   res 2                   ' the delimiter used in text input and a save location
base                    res 2                   ' current number base + backup location during overrides

lasttwo                 res 2                   ' last two characters (looking for sequences)
prompt                  res 2                   ' pointer to code to execute when Forth prompts for a new line
accept                  res 2                   ' pointer to code to execute when Forth accepts a line to interpret (0=ok)

spincnt                 res 1                   ' Used by spinner to rotate busy symbol
flags                   res 2

prefix                  res 1                   ' NUMBER prefix
suffix                  res 1                   ' NUMBER suffix

prevch                  res 1                   ' used to detect LF only sequences vs CRLF to perform auto CR
lastkey                 res 1
keychar                 res 1                   ' override for key character
keycnt                  res 1
endreg                  res 0



                                        { *** BYTECODE DEFINITIONS VECTOR TABLE *** }
{
Kernel bytecode definitions need to be called and this table makes it easy to do so
with just a 2 byte call. Extra memory may be allocated for user definitions as well
The Spin compiler requires longs whereas we only need 16-bit words but this will work anyway.
The runtime compiler can reuse the high-word of all these longs and compile a YCALL rather than
an XCALL so that the high-word is used instead.
Also another table has been added to expand the call vectors up to 1024 entries.

So there are 256 16-bits vectors x 4 pages using
XCALL      low word of first 1K page
YCALL      high word of first 1K page
ZCALL      low word of second 1K page
VALL       High word of second 1K page

}

DAT

                        org 0                   ' ensure references can be reduced to a single byte index to be called by XCALL xx
'
```

XCALLS

xXCALLS	long @_XCALLS+s
xSETQ	long @SETQ+s
xLT	long @LT+s
xZLT	long @ZLT+s
xULT	long @ULT+s
xWITHIN	long @WITHIN+s
xFILL	long @FILL+s
xERASE	long @ERASE+s
xms	long @ms+s
xus	long @us+s
xCOMMENT	long @COMMENT+s
xBRACE	long @BRACE+s
xCURLY	long @CURLY+s
xIFNDEF	long @IFNDEF+s
xIFDEF	long @IFDEF+s
xPRTHEX	long @PRTHEX+s
xPRTBYTE	long @PRTBYTE+s
xPRTWORD	long @PRTWORD+s
xPRTLONG	long @PRTLONG+s
xPRTSTK	long @PRTSTK+s
xPRTSTKS	long @PRTSTKS+s
xDEBUG	long @DEBUG+s
xDUMP	long @DUMP+s
xCOGDUMP	long @COGDUMP+s
xREBOOT	long @_REBOOT+s
xSTOP	long @STOP+s
xLOADMOD	long @aLOADMOD+s
xSSD	long @SSD+s
xESPIO	long @ESPIO+s
xSPIO	long @SPIO+s
xSPIOD	long @SPIOD+s
xSDRD	long @SDRD+s
xSDWR	long @SDWR+s
xPWM32	long @PWM32+s
xPWMST32	long @PWMST32+s
xPLOT	long @PLOT+s
xBCA	long @_BCA+s
'xCMPSTRMOD	long @CMPSTRMOD+s
xRCMOVE	long @RCMOVE+s
xCLS	long @CLS+s
xEMIT	long @EMIT+s
xSPACE	long @SPACE+s
xBELL	long @BELL+s
xCR	long @CR+s
xOK	long @OK+s
xSPINNER	long @SPINNER+s
xBIN	long @BIN+s
xDECIMAL	long @DECIMAL+s
xHEX	long @HEX+s
xREADBUF	long @READBUF+s
xCONKEY	long @CONKEY+s
xKEY	long @KEY+s
xWKEY	long @WKEY+s
xQEMIT	long @QEMIT+s
xTOUPPER	long @TOUPPER+s
xPUTCHAR	long @PUTCHAR+s
xPUTCHARPL	long @PUTCHARPL+s
xSCRUB	long @SCRUB+s
xdoCHAR	long @doCHAR+s
xGETWORD	long @GETWORD+s
xTICK	long @TICK+s
xATICK	long @ATICK+s
xNFATICK	long @NFATICK+s
x_NFATICK	long @_NFATICK+s
xTOVEC	long @TOVEC+s
xPFA	long @PFA+s
xIFEXIT	long @IFEXIT+s
xSEARCH	long @SEARCH+s
xFINDSTR	long @FINDSTR+s
'xCMPSTR	long @CMPSTR+s
xEXECUTE	long @EXECUTE+s
xGETVER	long @GETVER+s
xPRTVER	long @PRTVER+s
xTODIGIT	long @TODIGIT+s
xNUMBER	long @NUMBER+s
xTERMINAL	long @TERMINAL+s
xCONSOLE	long @CONSOLE+s
x_NUMBER	long @_NUMBER+s
xGRAB	long @GRAB+s
xRESFWD	long @RESFWD+s
xATPAD	long @ATPAD+s
xHOLD	long @HOLD+s
xTOCHAR	long @TOCHAR+s
xRHASH	long @RHASH+s
xLHASH	long @LHASH+s
xHASH	long @HASH+s
xHASHS	long @HASHS+s
x_STR	long @_STR+s
xPSTR	long @PSTR+s
xPRTSTR	long @PRTSTR+s
xSTRLEN	long @STRLEN+s
xUPRT	long @UPRT+s

```
xPRT                    long @PRT+s
xPRTDEC                 long @PRTDEC+s
xLITCOMP                long @LITCOMP+s
xBCOMP                  long @BCOMP+s
xBCOMPILE               long @BCOMPILE+s
xCOMPILE                long @COMPILE+s
xCCOMP                  long @CCOMP+s
xWCOMP                  long @WCOMP+s
xLCOMP                  long @LCOMP+s
xSTACKS                 long @STACKS+s
x_STR_                  long @_STR_+s
x_PSTR_                 long @_PSTR_+s
x_IF_                   long @_IF_+s
x_ELSE_                 long @_ELSE_+s
x_THEN_                 long @_THEN_+s
x_BEGIN_                long @_BEGIN_+s
x_UNTIL_                long @_UNTIL_+s
x_AGAIN_                long @_AGAIN_+s
x_REPEAT_               long @_REPEAT_+s
xMARK                   long @MARK+s
xUNMARK                 long @UNMARK+s
xVECTORS                long @VECTORS+s
xTASK                   long @TASK+s
xIDLE                   long @IDLE+s
xALLOT                  long @ALLOT+s
xALLOCATED                      long @ALLOCATED+s
xATO                    long @Ato+s
xATATR                  long @ATATR+s
xCOLON                  long @COLON+s
x_COLON                 long @_COLON+s
xPUBCOLON                       long @PUBCOLON+s
xPRIVATE                long @PRIVATE+s
xUNSMUDGE                       long @UNSMUDGE+s
xENDCOLON                       long @ENDCOLON+s
xCOMPILES               long @COMPILES+s
xCREATE                 long @CREATE+s
xCREATEWORD                     long @CREATEWORD+s
xCREATESTR                      long @CREATESTR+s
xAddACALL               long @AddACALL+s
xHERE                   long @_HERE+s
xNFACFA                 long @NFACFA+s
x_TACHYON               long @_TACHYON+s
xERROR                  long @ERROR+s
xNOTFOUND                       long @NOTFOUND+s
xDISCARD                long @DISCARD+s
xAUTORUN                long @AUTORUN+s
xFREE                   long @FREE+s
xAUTOST                 long @AUTOST+s
xFIXDICT                long @FIXDICT+s
xBUFFERS                long @BUFFERS+s
xCOLDST                 long @COLDST+s
xSWITCH                 long @_SWITCH+s
xSWITCHFETCH            long @SWITCHFETCH+s
xISEQ                   long @ISEQ+s
xIS                     long @IS+s
xISEND                  long @ISEND+s
xISWITHIN               long @ISWITHIN+s
xInitStack              long @InitStack+s
xSPSTORE                long @SPSTORE+s
xDEPTH                  long @_DEPTH+s
xCOGINIT                long @aCOGINIT+s

' NOTE: this table is limited to 1024 word entries but leave room for extensions and user application to use the rest of these

xLAST                   long 0[512-xLAST]                       ' Reserve the rest of the area possible
                        long 0                                  ' plus one extra for termination ?


                                                { *** HIGH LEVEL FORTH AREA ***  }

DAT

                                              { *** HIGH LEVEL BYTECODE DEFINITIONS *** }


' Emit a character
' EMIT ( char -- )
EMIT            byte    REG,uemit,WFETCH,QDUP,_IF,01,AJMP               ' execute user EMIT if vector is non-zero
                byte    XOP,pEMIT,EXIT

' Print inline string
PRTSTR          byte    RPOP
pstlp           byte    CFETCHINC,QDUP,_IF,04,XCALL,xEMIT,_AGAIN,@pstret-@pstlp
pstret          byte    PUSHR,EXIT
```

```
{ debug print routines - also used by DUMP etc }

' .HEX ( n -- ) print nibble n as a hex character
PRTHEX  ' ( n -- ) print n (0..$0F) as a hex character
                byte    toNIB
                byte    PUSH1,$30,PLUS
                byte    DUP,PUSH1,$39,GT,_IF,03,_BYTE,7,PLUS                    'Adjust for A..F
PRTCH           byte    XCALL,xEMIT,EXIT

 ' .BYTE ( n -- ) print n as 2 hex characters
PRTBYTE         byte    DUP,_4,_SHR
                byte    XCALL,xPRTHEX,XCALL,xPRTHEX,EXIT

 ' .WORD ( n -- ) print n as 4 hex characters
PRTWORD         byte  DUP,_8,_SHR
                byte  XCALL,xPRTBYTE
PW1             byte  XCALL,xPRTBYTE
PW2             byte  EXIT

 ' .LONG ( n -- ) print n as 8 hex characters
PRTLONG         byte  DUP,PUSH1,16,_SHR
                byte  XCALL,xPRTWORD
                byte  _BYTE,".",XCALL,xEMIT
PRL1            byte  XCALL,xPRTWORD
PRL2            byte  EXIT
' Deprecated in favour of BDUMP in EXTEND.fth - still used internally but renamed to HDUMP to avoid conflict
' DUMP ( addr cnt -- ) Hex dump of hub RAM - (NOTE: if CFETCH is vectored then other memory can be accessed)
DUMP            byte  ADO
                byte    XCALL,xCR
                byte    I,XCALL,xPRTWORD
                byte    XCALL,xPRTSTR,": ",0
                byte    I,PUSH1,$10,ADO
                byte      I,CFETCH,XCALL,xPRTBYTE
                byte      PUSH1,$20,XCALL,xEMIT,LOOP
                byte        XCALL,xPRTSTR,"   ",0
                byte        I,PUSH1,$10,ADO
                byte          I,CFETCH,DUP,BL,XCALL,xLT,OVER,PUSH1,$7E,GT,_OR
                byte          _IF,03,DROP,PUSH1,"."
                byte          XCALL,xEMIT,LOOP
                byte    PUSH1,$10,PLOOP
                byte    XCALL,xCR,EXIT

' COGDUMP ( addr cnt -- ) Dump cog memory, but try to minimize stack usage
COGDUMP         byte  REG,temp,WSTORE,REG,temp+2,WSTORE,JUMP,@cdm2-@cdmlp
cdmlp           byte      REG,temp+2,WFETCH,_3,_AND,ZEQ,_IF,@cdm3-@cdm2
cdm2            byte  XCALL,xCR,REG,temp+2,WFETCH,XCALL,xPRTWORD,XCALL,xPRTSTR,": ",0
cdm3            byte      REG,temp+2,WFETCH,XOP,pCOGFETCH,XCALL,xPRTLONG,BL,XCALL,xEMIT
                byte      _1,REG,temp+2,WPLUSST,MINUS1,REG,temp,WPLUSST
                byte      REG,temp,WFETCH,ZEQ,_UNTIL,@cdm1-@cdmlp
cdm1            byte      EXIT

PRTSTK
                byte      REG,base,CFETCH,PUSHR,XCALL,xDECIMAL
                byte      XCALL,xPRTSTR," Data Stack (",0,XCALL,xDEPTH,XCALL,xPRT,_BYTE,")",XCALL,xEMIT
                byte      XCALL,xDEPTH,_IF,@pstk2-@pstk3
pstk3           byte      _16,XOP,pCOGREG,XOP,pCOGFETCH,_8,PLUS,XCALL,xDEPTH,_SHL1,_SHL1,OVER,PLUS
pstklp          byte      XCALL,xCR,_BYTE,"$",XCALL,xEMIT,DUP,FETCH,DUP,XCALL,xPRTLONG
                byte      XCALL,xPRTSTR," - ",0,XCALL,xPRT
                byte      _4,MINUS,OVER,OVER,EQ,_UNTIL,@pstk1-@pstklp
pstk1           byte      DROP2
pstk2           byte      RPOP,REG,base,CSTORE,EXIT

STACKS          byte      _0,XOP,pCOGREG,_BYTE,tos-REG0,PLUS,EXIT   ' put the address of tos on the top of the stack

PRTSTKS     ' Print stacks but avoid cluttering with data from debug routines
                byte      XCALL,xPRTSTK
' RETURN STACK
                byte      XCALL,xSTACKS,PUSH1,datsz,PLUS,PUSH1,retsz
                byte      XCALL,xPRTSTR,$0D,$0A,"RETURN STACK ",0
                byte      XCALL,xCOGDUMP
' LOOP STACK
                byte      XCALL,xSTACKS,PUSH1,datsz+retsz,PLUS,PUSH1,loopsz
                byte      XCALL,xPRTSTR,$0D,$0A,"LOOP STACK ",0
                byte      XCALL,xCOGDUMP
' BRANCH STACK
                byte      XCALL,xPRTSTR,$0D,$0A,"BRANCH STACK ",0
                byte      XCALL,xSTACKS,PUSH1,datsz+retsz+loopsz,PLUS,PUSH1,branchsz
                byte      XCALL,xCOGDUMP
                byte      EXIT

' Print the stack(s) and dump the registers - also called by hitting <ctrl>D during text input
DEBUG           byte      XCALL,xPRTSTKS
                byte      XCALL,xPRTSTR,$0D,$0A,"REGISTERS",0
                byte      REG,temp,_WORD,1,00,XCALL,xDUMP
                byte      XCALL,xPRTSTR,$0D,$0A,"COMPILATION AREA",0
                byte      REG,here,WFETCH,PUSH1,$40,XCALL,xDUMP
                byte      EXIT




' COG CONTROL
```

```
                org
                byte    0                       ' align OPCODE long
_REBOOT
                byte    _BYTE,$80,OPCODE
                CLKSET  tos

                org
                byte    0,0                     ' align OPCODE long
STOP            ' STOP ( cog -- )
                byte    DROP,OPCODE             ' need to drop parameter before opcode which always EXITs but dropped is in X via POPX
                cogstop X


' HERE ( -- addr ) Address of next compilation location
_HERE           byte    REG,here,WFETCH,EXIT
```

{ TACHYON VM CODE MODULES }

```
' There are a number of longs reserved in the cog for a code module which can be loaded with LOADMOD and executed with RUNMOD

CON
loadsz          = 19                    ' Specifies the number of longs to load with LOADMOD for a RUNMOD module

DAT
                org $01F0-loadsz        ' fixed address - high-level code can always assume RUNMOD' cog origin is here
pRUNMOD
_RUNMOD         res loadsz
```

{ V2.4
Remove the dedicated CMPSTR instruction from the kernel and only load it as a PASM module when performing a full
source file load using TACHYON and END directives. So CMPSTR will be loaded with the TACHYON directive but otherwise
CMPSTR will be available as high-level function otherwise where compilation speed is not needed.
The TACHYON directive will also turn off echo and ok prompts as well as stop other TF cogs and keypoll/alarm functions.
}

```
{
                                org     _RUNMOD
' Write byte to I2C bus
' COGREGS: 0=scl 1=sda
' I2CWR ( send -- ack )  ( A: miso mosi sck )
_I2CWR                          mov     R1,#8
i2cwrlp                         andn    OUTA,sck                ' clock low
                                or      DIRA,mosi               ' make sure data is an output
                                shl     sdat,#1 wc              ' msb first
                                muxc    OUTA,mosi               ' send next bit of data out
                                call    #i2cdly
                                or      OUTA,sck                ' clock high
                                djnz    R1,#i2cwrlp
                                andn    OUTA,sck                ' get ready to read ack
                                andn    DIRA,mosi               ' float SDA
                                or      OUT,sck

                                jmp     unext
ic2dly                          nop
i2cdly_ret                      ret

I2CWR           byte    _WORD,(@_I2CWR+s)>>8,@_I2CWR+s,_BYTE,16,XOP,pLOADMOD,EXIT

}
```

{ *** **ENHANCED SERIAL PERIPHERAL INPUT OUTPUT MODULE** ***

- Transfers 1 to 32 bits msb first
- Transmit data does not need to be left justified to be ready for transmission
- Receive data is in correct format
- Data is shifted in and out while the clock is low
- The clock is left low between transfers unless a chip select mask is specified

}

```
                                org     _RUNMOD
' COGREGS: 0=sck 1=mosi 2=miso 3=cnt 4=cs
' ESPIO ( send -- receive )  ( A: miso mosi sck )
_ESPIO
                                andn    OUTA,scs                ' chip select low
                                mov     R1,#32
                                sub     R1,scnt
                                shl     tos,R1                  ' left justify transmit data
                                mov     R1,scnt
ESPIOlp                         andn    OUTA,sck                ' clock low
                                shl     sdat,#1 wc              ' assume msb first
                                test    miso,INA wz             ' test data from device while clock is low
                                muxc    OUTA,mosi               ' send next bit of data out
                        if_nz   or      sdat,#1                 ' now assemble data (also setup time for mosi)
                                or      OUTA,sck                ' clock high
                                djnz    R1,#ESPIOlp
                                tjnz    scs,#ESxt               ' leave with clock high if CS mask specified
                                andn    OUTA,sck                ' leave with clock low
ESxt                            or      OUTA,scs                ' chip select high
                                jmp     unext

ESPIO           byte    _WORD,(@_ESPIO+s)>>8,@_ESPIO+s
```

```
' this next line is common to other modules XCALL,xLOADMOD
aLOADMOD byte     _WORD,_RUNMOD>>8,_RUNMOD,_BYTE,loadsz,XOP,pLOADMOD,EXIT


                              org         _RUNMOD
' SSD module added for fast SSD2119 SPI operations
' COGREGS: 0=sck 1=mosi 2=miso 3=cnt 4=mode
' SSD ( send cnt --  )  ( A: miso mosi sck )
_SSD
SSDREP                        mov         X,tos+1
                              shl         X,#7                  ' left justify but leave msb zero (control byte)
                              call        #SSD8
                              call        #SSD8D
                              call        #SSD8D
                              djnz        tos,#SSDREP
                              jmp         #DROP2

SSD8D                         or          X,#1                  ' move 1 into msb = data
                              ror         X,#1
SSD8                          andn        OUTA,scs              ' chip select low
                              mov         R1,scnt
SSDlp                         andn        OUTA,sck              ' clock low
                              shl         X,#1 wc               ' assume msb first
                              muxc        OUTA,mosi             ' send next bit of data out
                              or          OUTA,sck              ' clock high
                              djnz        R1,#SSDlp
                              or          OUTA,scs              ' chip select high
SSD8D_ret
SSD8_ret                      ret

SSD           byte    _WORD,(@_SSD+s)>>8,@_SSD+s
              byte    XCALL,xLOADMOD,EXIT



{ *** SERIAL PERIPHERAL INPUT OUTPUT MODULE ***
Transfers 1 to 32 bits msb first
Transmit data must be left justified ready for transmission
Receive data is in correct format
Data is shifted in and out while the clock is low
The clock is left low between transfers

}
                              org         _RUNMOD
' COGREGS: 0=sck 1=mosi 2=miso 3=cnt
' SPIO ( send -- receive )  ( A: miso mosi sck )
_SPIO                         mov         R1,scnt
                              or          DIRA,mosi
SPIOlp                        andn        OUTA,sck              ' clock low
                              shl         sdat,#1 wc            ' assume msb first
                              test        miso,INA wz           ' test data from device while clock is low
                              muxc        OUTA,mosi             ' send next bit of data out
                    if_nz     or          sdat,#1               ' now assemble data (also setup time for mosi)
                              or          OUTA,sck              ' clock high
                              djnz        R1,#SPIOlp
                              andn        OUTA,sck              ' leave with clock low
                              jmp         unext

SPIO          byte    _WORD,(@_SPIO+s)>>8,@_SPIO+s
              byte    XCALL,xLOADMOD,EXIT



                              org         _RUNMOD
' COGREGS: 0=sck 1=mosi 2=miso 3=cnt
' COGREG4 = delay
' SPIOD ( send -- receive )  ( A: miso mosi sck )
_SPIOD                        mov         R1,scnt
                              or          DIRA,mosi
SPIODlp                       andn        OUTA,sck              ' clock low
                              shl         sdat,#1 wc            ' assume msb first
                              test        miso,INA wz           ' test data from device while clock is low
                              muxc        OUTA,mosi             ' send next bit of data out
                    if_nz     or          sdat,#1               ' now assemble data (also setup time for mosi)
                              or          OUTA,sck              ' clock high
                              mov         X,REG4
spiodly                       djnz        X,#spiodly
                              djnz        R1,#SPIODlp
                              andn        OUTA,sck              ' leave with clock low
                              jmp         unext

SPIOD         byte    _WORD,(@_SPIOD+s)>>8,@_SPIOD+s
              byte    XCALL,xLOADMOD,EXIT
```

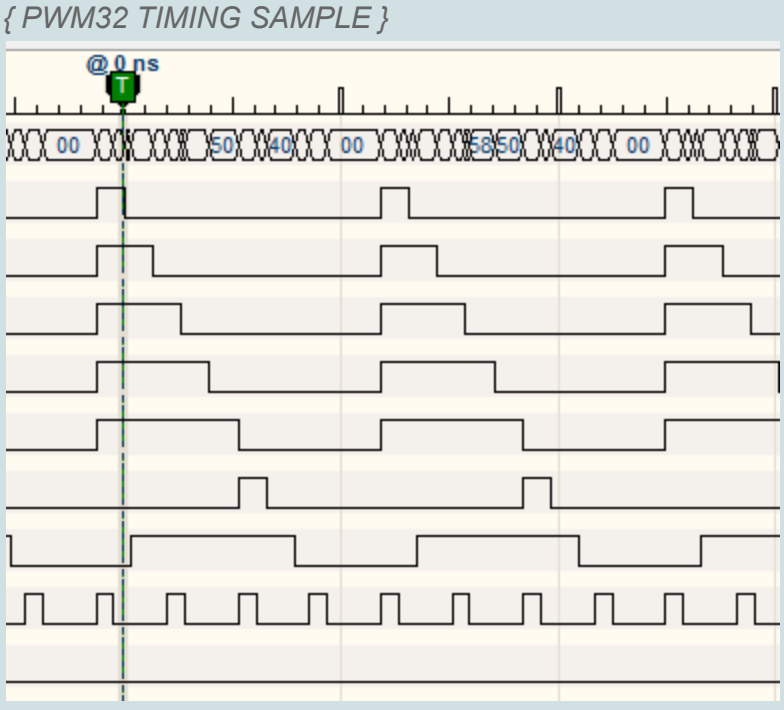
```
{ Deprecated for test scanning version
                        org     _RUNMOD
' Reads in a block from the SD card - 512 bytes takes 1.44ms or 2.92us/byte
' SDRD ( dst cnt --  ;read block from SD into memory )
_SDRD                   or      OUTA,mosi               ' keep data in high
                        or      DIRA,mosi
SDRDlp                  mov     R1,#8
SDRDbits                andn    OUTA,sck                ' clock low
                        test    miso,INA wc             ' test data from device while clock is low
                        rcl     R0,#1                   ' now assemble data (also setup time for mosi)
                        or      OUTA,sck                ' clock high
                        djnz    R1,#SDRDbits
                        andn    OUTA,sck                ' leave with clock low
                        wrbyte  R0,tos+1                ' write byte to destination
                        add     tos+1,#1                ' dst = dst+1
                        djnz    tos,#SDRDlp
                        jmp     #DROP2

' Load the SD read module (13us)
SDRD    byte    _WORD,(@_SDRD+s)>>8,@_SDRD+s
        byte    XCALL,xLOADMOD,EXIT

}

' SDRD ( 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
' NOTE: ensure MOSI is set as an output high from caller by  1 COGREG@ OUTSET
                        org     _RUNMOD
_SDRD
                        mov     X,tos+1                 ' dst --> X
                        mov     R2,dst1 '$200           ' R2 = 512 bytes
SDRDSClp                        mov                     R1,#8           ' 8-bits
SDRDSCbits              andn    OUTA,sck                ' clock low
                        test    miso,INA wc             ' test data from device while clock is low
                        rcl     R0,#1                   ' now assemble data (also setup time for mosi)
                        or      OUTA,sck                ' clock high
                        djnz    R1,#SDRDSCbits
                        andn    OUTA,sck                ' leave with clock low
                        wrbyte  R0,X                    ' write byte to destination
                        and     R0,#$FF                 ' only want to compare a byte value
                        cmp     R0,tos wz               ' compare byte with char unsigned for equality
                if_z    add     ACC,#1                  ' found one and increment count
                if_z    cmp     ACC,#1 wz               ' if match and count = 1 (first occurrence)
            if_z        mov     tos+1,X                 ' then save dst (tos+2) to firstpos (tos)
                        add     X,#1                    ' dst = dst+1
                        djnz    R2,#SDRDSClp            ' next
                        call    #POPX                   ' discard "char"
                jmp     #PUSHACC                        ' push char count and clear ACC

' Load the SDSCAN read module (13us)
SDRD    byte    _WORD,(@_SDRD+s)>>8,@_SDRD+s
        byte    XCALL,xLOADMOD,EXIT

{
' moved to SDCARD.fth MJB
        WORD scancnt,scanpos
        BYTE scanch
        --- Read in one block of 512 bytes from SD to dst
        pri (SDRD) ( dst -- )
                        \ BYTE scanch holds the char to scan for
                        \ gives number of character matches found in WORD scancnt
                        \ autoincrements on each call, use scancnt W~ to init to 0 if needed
                        \ position of first match in WORD scanpos
                [SDRD] 1 COGREG@ OUTSET scanch C@ RUNMOD
                scancnt W+! scanpos W@ 0= IF scanpos W! ELSE DROP THEN   ' to catch the earliest over many block reads - use scanpos W~ to init
        ;
}

' SD MODULE
                        org     _RUNMOD
' Write a block to the SD card - 512 bytes
' SDWR ( src cnt --  ;write block from memory to SD )
_SDWR
                        or      DIRA,mosi
SDWRlp                  rdbyte  R0,tos+1                ' read next byte from source
                        add     tos+1,#1                ' increment source address
                        shl     R0,#24                  ' left justify
                        mov     R1,#8                   ' send 8 bits
SDWRbits                andn    OUTA,sck                ' clock low
                        shl     R0,#1 wc                ' assume msb first
                        muxc    OUTA,mosi               ' send next bit of data out
```

```
                        or      OUTA,sck                ' clock high
                        djnz    R1,#SDWRbits
                        andn    OUTA,sck                ' leave with clock low
                        djnz    tos,#SDWRlp             ' next byte
                        jmp     #DROP2

' Load the SD write module
SDWR    byte    _WORD,(@_SDWR+s)>>8,@_SDWR+s
        byte    XCALL,xLOADMOD,EXIT
```

{ * 32 channel 8-bit PWM *** }**

```
' [PWM32] loads the PWM32 module into the cog ready for a RUNMOD

PWM32   byte    _WORD,(@_PWM32+s)>>8,@_PWM32+s
        byte  XCALL,xLOADMOD,EXIT


' kuroneko's 7.6kHz version of the 32 channel PWM module

                org     _RUNMOD             ' Compile for RUNMOD area in cog
_PWM32          movi    ctrb, #%0_11111_000     ' LOGIC.always
                mov     frqb, tos+1             ' table base address
                shr     frqb, #1{/2}            ' divided by 2 (expected to be 4n)
                mov     cnt,#5{14}+2*4+23       ' minimal entry delay
                add     cnt,cnt
                mov     phsb,#0                 ' preset (not optional, 8n)

pwmlp32         and     phsb,wrap32             ' $0400 to $0000 transition     |
                rdlong  X,phsb                  ' read from index + 2*table/2   | need to be back-to-back

                waitcnt cnt,tos                 ' |
                mov     outa,X                  ' update outputs

                add     phsb,#4                 ' update read ptr               |
                rdlong  X,phsb                  ' read from index + 2*table/2   | need to be back-to-back

                add     phsb,#4                 ' update read ptr
                waitcnt cnt,tos                 ' |
                mov     outa,X                  ' update outputs

                jmp     #pwmlp32

wrap32          long    256 * 4 -1
```

{ PWM32 TIMING SAMPLE }



' PWM32! SETUP TABLE MODULE

```
' this module is run from the controlling cog whereas the PWM32 runtime is in a dedicated cog
                                org     _RUNMOD
' _PWM32! ( duty8 mask table -- )                ' 104us (LOADMOD takes 18.8us)
_PWMST32        mov     R2,#256                 ' scan through all 256 slices
                add     tos,endtbl              ' start from end (optimize hub access)
                add     tos+2,#1                ' compensate for cmp op so that $80 = 50 %, $100 = 100 %
pwmstlp32       rdlong  X,tos                   ' read one 1/256 slice of 32 channels
                cmp     R2,tos+2 wc             ' is duty8 > R2
                muxc    X,tos+1                 ' set or clear the bit(s)
                wrlong  X,tos                   ' update slice
```

```
                sub     tos,#4              ' next long
                djnz    R2,#pwmstlp32       ' terminate on wrap
                jmp     #DROP3
endtbl            long    256 * 4 -4        ' offset to last long of table

' [PWM32!]  Load the PWM32! module which is used to setup the PWM table
PWMST32   byte   _WORD,(@_PWMST32+s)>>8,@_PWMST32+s
        byte  XCALL,xLOADMOD,EXIT



' PLOT MODULE
' Used for VGA/TV or LCD graphics
' pixshift is always a multiple of two, 512 pixels/line = 6 etc
'
                        org     _RUNMOD
' PLOT ( x y -- )
_PLOT                   shl     tos,pixshift            ' n^2 bytes/Y line
                        mov     X,tos+1
                        shr     tos+1,#3                ' byte offset in line
                        add     tos,tos+1               ' byte offset in frame
                        add     tos,pixeladr            ' byte address in memory
                        and     X,#7                    ' get bit mask
                        mov     tos+1,#1
                        shl     tos+1,X
                        jmp     #SET
' [PLOT]
PLOT      byte    _WORD,(@_PLOT+s)>>8,@_PLOT+s
          byte    XCALL,xLOADMOD,EXIT



{

CLKSET
(c) Copyright 2009 Philip C. Pilgrim (propeller@phipi.com)
see end of file for terms of use.

This object determines whether a 5MHz or 10MHz crystal is
being used and sets the PLL accordingly for 80MHz operation.
The main program should use the following settings:
  _clkmode      = xtal1 + pll8x
  _xinfreq      = 10_000_000
}
          org     _RUNMOD

_setpll     movi    ctra,#%0_00001_011      'Set ctra for pll on no pin at x1.
          movi    frqa,#%0100_0000_0      'Set frqa for clk / 4 (= 20MHz w/ 10MHz crystal, i.e. too high).
          add     pllx,cnt                'Give PLL time to lock (if it can) and stabilize.
          waitcnt  pllx,#0
          movi    vcfg,#$40               'Configure video for discrete pins, but no pins enabled.
          mov     vscl,#$10               'Set for 16 clocks per frame.
          waitvid  0,0                    'Wait once to clear time.
          neg     pllx,cnt                'Read -cnt.
          waitvid  0,0                    'Wait 16 pll clocks = 64 processor clks.
          add     pllx,cnt                'Compute elapsed time.
          cmp     pllx,#$40 wz            'Is it really 64 clocks?
    if_z  mov     pllx,#$6f               '  Yes: Crystal is 5MHz. Set clock mode to XTAL1 and PLL16X.
    if_z  clkset  pllx
    if_z  wrbyte   $-2, #4                'update clkmode location
                jmp     unext

pllx        long    $1_0000               '65536 clks for pll to stabilize.


'( BYTE CODE ANALYSER - 140606 - temporary, testing bytecode frequencies with MJB )

                        org     _RUNMOD
' bytecode analyser
BCARUN                  mov     unext,bcavec
bca
                        rdbyte  instr,IP                ' read byte code instruction
                        add     IP,#1 wc                'advance IP to next byte token (clears the carry too!)
                        mov     R0,instr                ' only process single byte opcodes
                        shl     R0,#2                   ' address longs
                        add     R0,bcabuf               ' in bcabuf (1K at BUFFERS)
                        rdlong  X,R0                    ' increment the counter for this bytecode
                        add     X,#1
                        wrlong  X,R0
                        jmp     instr                   'execute the code by directly indexing the first 256 long in cog

bcavec          long bca
bcabuf          long @RESET+s                   ' use BUFFERS (normally at $7500) buffers uses kernel image from RESET
```

```
' Load the bytecode analyser module
_BCA        byte    _WORD,(@bcarun+s)>>8,@bcarun+s ' fixed to point to bcarun not bca
            byte    XCALL,xLOADMOD,XOP,pRUNMOD,EXIT



' !SP - Initialize the data stack to a depth of zero
InitStack
            byte    _0,_BYTE,depth-REG0,XOP,pCOGREG,XOP,pCOGSTORE,EXIT zero the depth index

' SP! ( addr -- )
' Assign a data stack pointer for this cog - depth depends on use - typically 8 to 32 longs required
SPSTORE     byte    _BYTE,stkptr-REG0,XOP,pCOGREG,XOP,pCOGSTORE,EXIT

' DEPTH ( -- levels )
_DEPTH      byte    _BYTE,depth-REG0,XOP,pCOGREG,XOP,pCOGFETCH,_SHR1,_SHR1,DEC,EXIT

' BUFFERS ( -- addr )
BUFFERS     byte    _WORD,(@RESET+s)>>8,@RESET+s,EXIT
```

Table 3-1: Destination Register Fields			
31:18	17:4	3	2:0
14-bit Long address for PAR Register	14-bit Long address of code to load	New	Cog ID

```
' COGINIT ( code pars cog -- ret )
aCOGINIT    byte    SWAP,_4,_SHL,_OR,SWAP,_BYTE,18,_SHL,_OR,XOP,_COGINIT,EXIT
```

{*** NUMBER PRINT FORMATTING ***}

```
' @PAD ( -- addr ) pointer to current position in number pad
ATPAD       byte    REG,padwr,CFETCH,REG,numpad,PLUS,EXIT

' HOLD ( char -- )
HOLD        byte    MINUS1,REG,padwr,CPLUSST,XCALL,xATPAD,CSTORE,EXIT

' >CHAR  ( val -- ch ) convert binary value to an ASCII character
TOCHAR      byte    PUSH1,$3F,_AND,PUSH1,"0",PLUS,DUP,PUSH1,"9"             ' convert to "0".."9"
            byte    GT,_BYTE,7,_AND,PLUS                                      ' convert to "A"..
            byte    DUP,PUSH1,$5D,GT,ZEXIT,_3,PLUS,EXIT                     ' skip symbols to go to "a"..

' #> ( n1 -- caddr )
RHASH       byte    DROP,XCALL,xATPAD,_BYTE,leadspaces,REG,flags,CLR,EXIT


' <#          ' resets number pad write index to end of pad
LHASH       byte    PUSH1,numpadsz,REG,padwr,CSTORE,_0,XCALL,xHOLD
            byte EXIT

' # ( n1 -- n2 ) convert the next ls digit to a char and prepend to number string
HASH        byte    DUP,_IF,@has1-@has2
has2        byte    REG,base,CFETCH,UDIVMOD,SWAP,XCALL,xtoCHAR
            byte    XCALL,xHOLD,EXIT
            ' conversion digits exhausted, use zeros or spaces
has1        byte    _BYTE,$30,_BYTE,leadspaces,REG,flags,XCALL,xSETQ,_IF,02,DROP,BL,XCALL,xHOLD,EXIT

' #S ( n1 -- 0 ) Convert all digits
HASHS       byte    XCALL,xHASH,DUP,ZEQ,_UNTIL,06,EXIT

' STREND ( str -- strend )
'STREND     byte    CFETCHINC,QDUP,_IF,06,DUP,_BYTE,$7E,GT,_UNTIL,10,EXIT

STRLEN      ' ( str -- len )
            byte    DUP,STREND,SWAP,MINUS,EXIT

' STR ( -- n ) Leave address of inline string on stack and skip to next instruction
_STR        byte    RPOP,DUP
STRlp       byte    CFETCHINC,ZEQ,_UNTIL,04,PUSHR,EXIT

' . ( n -- ) Print the number off the stack
PRT         byte    DUP,XCALL,xZLT,_IF,05,PUSH1,$2D,XCALL,xEMIT,NEGATE
' U. ( n -- ) Print an unsigned number
UPRT        byte    XCALL,xLHASH,XCALL,xHASHS,XCALL,xRHASH
' .STR ( adr -- ) Print the null or 8th bit terminated string
PSTR        byte    CFETCHINC,DUP,ZEQ,OVER,_BYTE,$7F,GT,_OR,_IF,02,DROP2,EXIT,XCALL,xEMIT,_AGAIN,16



PRTDEC      byte    _BYTE,10
```

```
' DEPPRECATED - only used by END - use .NUM from EXTEND.fth
' B. ( n base -- ) Print the number off the stack in the base specified
basePRT  byte    REG,base,CFETCH,PUSHL,REG,base,CSTORE
         byte    XCALL,xLHASH,XCALL,xHASH,XCALL,xHASH,XCALL,xHASH,XCALL,xHASHS,XCALL,xRHASH,XCALL,xPSTR
         byte    LPOP,REG,base,CSTORE,EXIT
```

{ *** OPERATORS *** }

```
' 0< ( n -- flg )
ZLT      byte  _0,XCALL,xLT,EXIT

' < ( n1 n2 -- flg )
LT       byte  SWAP,GT,EXIT

' U<
ULT      byte  OVER,OVER,_XOR,XCALL,xZLT,_IF,04
         byte  NIP,XCALL,xZLT,EXIT
         byte  MINUS,XCALL,xZLT,EXIT

' ( n lo hi -- flg ) true if n is within range of low and high inclusive
WITHIN   byte    INC,OVER,MINUS,PUSHR
         byte    MINUS,RPOP,XCALL,xULT
WT1      byte    ZEQ,ZEQ,EXIT

' SET? ( mask caddr -- flg ) Test single bit of a byte in memory
SETQ     byte    CFETCH,_AND,ZEQ,ZEQ,EXIT

' ?EXIT ( flg -- ) Exit if flg is true
IFEXIT   byte    _IF,02,RPOP,DROP,EXIT

" The read and write index is stored as two words preceding the buffer, read this as a word (faster)
" BKEY ( buffer -- ch ) ' byte size buffer is preceded with a read index, go and read the next character
'
' READBUF ( buffer -- ch|$100 )
READBUF
         byte    DUP,DEC,DEC,DUP,WFETCH                  ' point to read index ( buffer writeptr writeindex )
         byte    SWAP,DEC,DEC,WFETCH,SWAP                ' ( buffer readindex writeindex )
         byte    OVER,EQ,_IF,03,DROP2,_0,EXIT            ' empty, return with null
         ' ( buffer read )
         byte    OVER,PLUS,CFETCH                                  ' get character from buffer
' perform auto LF to CR subs (but not when it is part of a CRLF )
         byte    DUP,_BYTE,$0A,EQ
         byte    REG,prevch,CFETCH,_BYTE,$0D,EQ,ZEQ,_AND
         byte    _3,_AND,PLUS                            ' convert the LF to a CR
'
         byte DUP,REG,prevch,CSTORE
         byte _WORD,$01,00,PLUS                          ' get char ( buffer [buffer+read]+$100 )
         byte    SWAP,_4,MINUS                           ' key readptr )
         byte    DUP,WFETCH,INC                          ' update read index and wrap
         byte    REG,rxsz,WFETCH,DEC,_AND,SWAP,WSTORE
         byte    EXIT

{
' Check the serial input stream for a key
' KEY? ( -- ch flg )
KEYQ
         byte    XCALL,xKEY,DUP,toBYTE,SWAP,EXIT         ' uses standard KEY word which always returns with 0 if no key

         byte    REG,ukey,WFETCH,QDUP,_IF,03,ACALL,DUP,EXIT          ' execute user code if set
         byte    REG,rxptr,WFETCH,XCALL,xREADBUF         ' read a char from the buffer
         byte    DUP,toBYTE,SWAP,_8,_SHR,ZEQ,ZEQ         ' convert to char and flag format
         byte    DUP,ZEQ,ZEXIT                           ' skip user keypoll if we are busy with a key
         byte    REG,keypoll,WFETCH,QDUP,_IF,01,ACALL    ' do a user keypoll if nothing else is happening
         byte    EXIT
}
{ always best to look at the need - what drives it - and then how well it addresses the need - example?
might have to come back a little later -- in the hour...

about I/O Streams
the Tachyon interpreter loops input and output can be revectored to:
1. the console (serial) with
        inputStream   KEY    &   ukey = 0
        outputStream  Emit   &   uEmit = 0
2. now we have
            LAN (actually 4/8 sockets)  with both input & output streams
3. we could have/have?  Bluetooth i/o
4. we have File i/o   FIMPUT / FOUTPUT
5. I created a STRING input stream 'class' where ukey can be revectored to e.g. to evaluate a STRING
            there is no corresponding string output stream yet, but could be easily created
6. I created a BUFFER input stream, which is similar to the STRING input stream, but reads from any user defined buffer
            actually the standard keyboard input stream is s.th. very similar reading from the console input buffer.
```

my buffer input stream currently does NOT handle buffer wrap around as would be needed for a ringbuffer
like the console input buffer.
7. I created a kind of filter stream, which sits on top of any other input stream and performs some
modifications. here especially transforming URL escape codes (the %20 etc.) back to the respective ASCII char
(in EVAL$.FTH shared with you)
btw.: similar to a STRING$ 'class' we could have a BUFFER§ class with read&write pointers and size just before the buffer area
and standard methods for reading/writing a buffer val bufAdr §! and bufAdr §@ -> val
e.g. what you have as the console input buffer ... , and of course the SD buffer SDBUF with val SDBUF §!
pub §! (val bufadr --) 2- DUP ROT SWAP W@ C! W++ (+ handle wrap) ;
pub §@ (bufaddr -- val) 2- 2- DUP W@ SWAP W++ (+ handle wrap) ;
MAKEBUF (size <name>) creates the buffer allocating 3 words for (rd/wr ptr + size) + size bytes
such a buffer would be the ideal base for a read/write stream.
btw: **READBUF** ~~**would not need the special regs to store sz&ptr if handled as**~~ **a generic buffer - it is needed, before the generic BUFFER is there so forget this**
a MAKEBUFSTREAM (size <bufname>) would perform a MAKEBUF and create 2 additions words
pub <bufname>RD and <bufname>WR which can be used as ukey / uemit values.
pub <bufname>RD (bufadr -- val) ' <bufname> §@ ;

I am not sure I like the new KEY behaviour with respect to a uniform way of reading all the different types of input stream, as described above ...

btw: in your assembler the # immediate char could also be handled by the unknown word hook ??
to make it even more PASM like (not that it is really required ;-)

\ just watching I was using the ukey redirection as well for EVAL$ etc. as a StringReadStream
\ did you see it? having a more unified read/write stream mechanism might be useful.
currently there are many small pieces

```
}
' PUTKEY ( ch -- ) Force a character as the next KEY read
' PUTKEY          byte        REG,keychar,CSTORE,EXIT
' V2.4 relegates KEY to a character stream so therefore nulls are not passed but a null is a null, same as no key
' KEY ( -- ch ) if ch is zero then no key was read
KEY
            byte    REG,keychar,CFETCH,QDUP,_IF,06,_0,REG,keychar,CSTORE,JUMP,@CHKKEY-@ky00 ' read a "key" that was forced with KEY!
ky00
            byte    REG,ukey,WFETCH,QDUP                        ' or be redirected to a user key routine?
            byte    _IF,03,ACALL,JUMP,@CHKKEY-@ky01             ' redirect key input to ukey vector
ky01                                                            ' Default key input if ukey is not set
CONKEY
            byte    REG,rxptr,WFETCH,XCALL,xREADBUF             ' this returns with a character with b8 = $100 ? set or a false
            byte    DUP,_IF,03,toBYTE,JUMP,@CHKKEY-@DOPOLL      ' return now if we have a key striped back to 8-bits w/o background polling
                                                                ' background polling while waiting for a key
DOPOLL      byte    REG,keypoll,WFETCH,QDUP,_IF,01,ACALL,EXIT   ' execute background polling while waiting for input

' WKEY ( -- ch ) wait for a key and return with character
WKEY        byte    XCALL,xKEY,QDUP,_UNTIL,05,toBYTE,EXIT

' keep a track of the position of the this key on the input line (useful for assembler etc)
CHKKEY      byte    _1,REG,keycnt,CPLUSST,DUP,_BYTE,$0D,EQ,ZEXIT,_0,REG,keycnt,CSTORE,EXIT
```

{ *** COMMENTING *** }

```
'           \       ( -- )
'           Ignore following text till the end of line.
'           IMMED
COMMENT
            byte    REG,delim+1,CFETCH,_BYTE,$0D,EQ,ZEQ,ZEXIT
            byte    XCALL,xKEY,_BYTE,$0D,EQ,_UNTIL,7            ' terminate comment on a CR
            byte    _BYTE,$0D,REG,keychar,CSTORE,EXIT           ' force a CR back into the key stream on exit
'           (       stack or other short inline comments )
BRACE       byte    XCALL,xWKEY,DUP,XCALL,xQEMIT,_BYTE,")",EQ,_UNTIL,10,EXIT

IFDEF       byte    XCALL,xNFATICK,ZEQ,JUMP,02
IFNDEF      byte    XCALL,xNFATICK,ZEXIT
' {         Block comments - allow nested {{  }} operation
CURLY       byte    _1,REG,11,CSTORE                                ' allow nesting by counting braces
CURLYlp     byte    XCALL,xWKEY                                     ' keep reading each char until we have a matching closing brace
            byte DUP,_BYTE,"{",EQ,_IF,04,_1,REG,11,CPLUSST      ' add up opening braces
            byte _BYTE,"}",EQ,_IF,04,MINUS1,REG,11,CPLUSST      ' count down closing braces
            byte REG,11,CFETCH,ZEQ,_UNTIL,@CURLYxt-@CURLYlp
CURLYxt     byte EXIT
```

{ *** MOVES & FILLS *** }

```
' <CMOVE ( src dst cnt -- ) byte move in reverse from the ends to the start
RCMOVE      byte    ROT,OVER,PLUS,DEC,ROT,THIRD,PLUS,DEC,ROT,XOP,pRCMOVE,EXIT

' ( addr cnt -- )
ERASE       byte    _0
```

```
' ( addr cnt fillch -- )
FILL        byte    THIRD,CSTORE,DEC,OVER,INC,SWAP,XOP,pCMOVE,EXIT
```

{ *** TIMING *** }

```
' ms ( n -- ) Wait for n milliseconds
ms          byte  QDUP,ZEXIT,PUSH3,$01,$38,$80,UMMUL,DROP,XOP,pDELTA,EXIT
' us ( n -- ) Wait for n microseconds (note- not accurate for small amounts)
us          byte  QDUP,ZEXIT,PUSH1,80,UMMUL,DROP,XOP,pDELTA,EXIT
```

{ *** NUMBER BASE *** }

```
' change the default number bases
BIN         byte  _2,JUMP,@SetBase-@DECIMAL
DECIMAL     byte  PUSH1,10,JUMP,@SetBase-@HEX
HEX         byte  PUSH1,16
SetBase     byte  REG,BASE,CSTORE,EXIT
```

{ *** OUTPUT OPERATIONS *** }

```
CLS         byte    PUSH1,$0C,XCALL,xEMIT,EXIT
SPACE       byte    BL,XCALL,xEMIT,EXIT
BELL        byte    _BYTE,7,XCALL,xEMIT,EXIT
CR          byte    PUSH1,$0D,XCALL,xEMIT,PUSH1,$0A,XCALL,xEMIT,EXIT
SPINNER     byte    REG,spincnt,CFETCH,_3,_SHR,_3,_AND,XCALL,x_STR,"|/-\",0,PLUS,CFETCH
            byte    XCALL,xEMIT,_8,XCALL,xEMIT,_1,REG,spincnt,CPLUSST,_1,XCALL,xms,EXIT


' PROMPT
OK          byte  XCALL,xPRTSTR," ok",$0D,$0A,0,EXIT


' ?EMIT      ,( ch --  ) suppress emitting the character if echo flag is off
QEMIT       byte _BYTE,echo,REG,flags,XCALL,xSETQ,_IF,03,XCALL,xEMIT,EXIT,DROP,EXIT


TULP        byte  INC
' >UPPER  ( str1 --  ) Convert lower-case letters to upper-case
TOUPPER
            byte  DUP,CFETCH,QDUP,_IF,@TUX-@TU1                 ' end of string?
TU1         byte  _BYTE,"a",_BYTE,"z",XCALL,xWITHIN
            byte  _UNTIL,@TU2-@TULP
TU2         byte  _BYTE,-$20,OVER,CPLUSST,_AGAIN,@TUX-@TULP                 ' convert case
TUX         byte  DROP,EXIT
```

{ *** STRING TO NUMBER CONVERSION *** }

```
' functional test for now - optimize later
' Convert ASCII value as a digit to a numeric value - only interested in bases up to 16 at present
'
TODIGIT     ' ( char -- val true | false )
            byte    DUP,PUSH1,"0",PUSH1,"9",XCALL,xWITHIN,_IF,@td8-@td7         ' only work with 0..9,A..F
td7         byte    PUSH1,"0",MINUS,_TRUE,EXIT                                 ' pass decimal digits
td8         byte    DUP,PUSH1,"A",PUSH1,"F",XCALL,xWITHIN,_IF,@td2-@td1
td1         byte    PUSH1,$37,MINUS,_TRUE,EXIT                                 ' pass hex digits
td2         byte    DROP,_FALSE,EXIT

{  Try to convert a string to a number
Allow all kinds of symbols but these are the rules for it to be treated as a number.
1. Leading character must be either a recognized prefix or a decimal digit
2. If trailing character is a recognized suffix then the first character must be a decimal digit
Acceptable forms are:
$1000       hex number
1000h
#1000       decimal number
1000d
%1000       binary number
1000b
```

Also as long as the first character and last character are valid (0..9,prefix,suffix) then any symbols me be mixed in the number i.e.
11:59 11.59 #5_000_000
}

```
_NUMBER ' ( str -- value digits | false )
            byte    _0,REG,4,STORE                                  ' REG0L = 0
            byte    _BYTE,sign,REG,flags,CLR                        ' clear sign
snlp        byte    DUP,CFETCH,REG,prefix,CSTORE                    ' save prefix (it may or may not be)
            byte  DUP,STREND,DEC,CFETCH,REG,suffix,CSTORE           ' save suffix (assume string has count byte)
            byte    DUP,CFETCH,_BYTE,"-",EQ,_IF,@sn01-@sn00         ' save SIGN
sn00        byte    _BYTE,sign,REG,flags,SET,INC
sn01
            ' PREFIX HANDLER
            byte    DUP,CFETCH                                      ' check prefix
            '       ( str ch )
            byte    _FALSE                                          ' preset prefix flag = false
            byte    OVER,PUSH1,"$",EQ,_IF,04,XCALL,xHEX,_TRUE,_OR   ' as does $ - also set hex base
            byte    OVER,PUSH1,"#",EQ,_IF,04,XCALL,xDECIMAL,_TRUE,_OR ' as does # - also set decimal base
            byte    OVER,PUSH1,"%",EQ,_IF,04,XCALL,xBIN,_TRUE,_OR   ' as does % - also set binary base
            byte    OVER,PUSH1,"&",EQ,_IF,@pf1-@pf0                 ' as does & - also set decimal base and IP notation
pf0         byte    XCALL,xDECIMAL,_TRUE,_OR
            byte    _BYTE,$80,REG,bnumber+3,CSTORE                  ' this forces "." symbols to work the same as ":"

pf1         byte    DUP,_IF,04,ROT,INC,ROT,ROT                      ' adjust string pointer to skip prefix
            '       ( str ch flg )
            byte    SWAP,PUSH1,"0",PUSH1,"9",XCALL,xWITHIN,_OR      ' 0..9 forces processing as a number
            '       ( str flg )
            byte    ZEQ,_IF,03,DROP,_FALSE,EXIT                     ' Give up now, it isn't a candiate
            '       ( str )                                         ' so far, so good, now check suffix
            ' SUFFIX HANDLER
            byte    REG,suffix,CFETCH
            byte    DUP,PUSH1,"0",PUSH1,"9",XCALL,xWITHIN           ' 0..9
            byte    OVER,PUSH1,"A",PUSH1,"F",XCALL,xWITHIN,_OR      ' A..F ( str sfx flg ) true if still a digit
            byte    OVER,PUSH1,"h",EQ,_IF,04,XCALL,xHEX,_TRUE,_OR   ' h = HEX
            byte    OVER,PUSH1,"b",EQ,_IF,04,XCALL,xBIN,_TRUE,_OR   ' b = BINARY
            byte    SWAP,PUSH1,"d",EQ,_IF,04,XCALL,xDECIMAL,_TRUE,_OR ' d = DECIMAL
            byte    ZEQ,_IF,03,DROP,_FALSE,EXIT                     ' bad suffix, no good
            ' so far the prefix and suffx have been checked prior to attempt a number conversion
            ' From here on there must be at least one valid digit for a number to be accepted
            ' DIGIT EXTRACTION & ACCUMULATION
nmlp        byte    DUP,CFETCH,DUP,_IF,@nmend-@nm1                  ' while there is another character
nm1         byte    XCALL,xTODIGIT,_IF,@nmsym-@nm2                  ' convert to a digit? or else check symbol
nm2         ' a digit has been found but is it valid for this base?         ' ( str val )
            byte    DUP,REG,BASE,CFETCH,DEC,GT,_IF,@nmok-@nm3
nm3         byte    DROP2,_FALSE,EXIT                                 ' a digit but exceeded base
nmok        byte    REG,anumber,FETCH,REG,BASE,CFETCH,UMMUL,DROP      ' shift anumber left one digit (base)
            byte    PLUS,REG,anumber,STORE                          ' and merge in new digit
            byte    _1,REG,digits,CPLUSST                           ' update number of digits
nmnxt       byte    INC,_AGAIN,@nmsym-@nmlp                         ' update str and loop

            ' character was not a digit - check for valid symbols (keep it simple for now)
            ' SYMBOLS
nmsym       byte    DUP,CFETCH,_BYTE,":",EQ
            byte    OVER,CFETCH,_BYTE,".",EQ
            byte    DUP,_IF,06,REG,digits,CFETCH,REG,dpl,CSTORE     ' remember last decimal place
            byte    REG,bnumber,FETCH,ZEQ,ZEQ,_AND,_OR
            byte    _IF,@nmsym2-@nmsym1                             ' Use : as special byte shift for IP notation etc
nmsym1      byte    REG,bnumber,FETCH
            byte    REG,anumber,FETCH,PLUS,_8,_SHL
            byte    REG,bnumber,STORE,_0,REG,anumber,STORE          ' accumulate & number in bnumber
nmsym2      byte    _AGAIN,@nmend-@nmnxt                            ' just ignore other symbols for now
            '
nmend       ' end of string - check
            byte    DROP2,REG,digits,CFETCH,DUP,ZEXIT               ' return with false if there are no digits
            byte    REG,anumber,FETCH,REG,bnumber,FETCH,PLUS
            byte    _BYTE,sign,REG,flags,XCALL,xSETQ,QNEGATE
            byte    SWAP,EXIT                                       ' all good, return with number and true

' NUMBER processing -try to convert a string to a number
NUMBER      ' ( str -- value digits | false )
            byte    DUP,XCALL,xSTRLEN,_2,EQ                         ' process control prefix i.e. ^A
            byte    OVER,CFETCH,_BYTE,"^",EQ,_AND,_IF,@ch01-@ctlch  ' ^ch  Accept caret char as <control> char
ctlch       byte    INC,CFETCH,_BYTE,$1F,_AND,_1,EXIT               ' control character processed

ch01        byte    DUP,XCALL,xSTRLEN,_3,EQ                         ' process character literal i.e. "A"
            byte    OVER,CFETCH,_BYTE,$22,EQ,_AND,_IF,@ch02-@ascch  ' "ch" Accept as an ASCII literal
ascch       byte    INC,CFETCH,_1,EXIT


                                                                    ' It wasn't an ASCII literal, process as a number
ch02        byte    REG,anumber,_BYTE,10,_0,XCALL,xFILL             ' zero out assembled number (double), digits, dpl
            byte    REG,base,CFETCH,REG,base+1,CSTORE               ' backup current base as it may be overridden
            byte    XCALL,x_NUMBER
```

```
nmb1        byte    REG,base+1,CFETCH,REG,base,CSTORE,EXIT                  ' restore default base before returning
```

{ *** COMPILER EXTENSIONS *** }

```
' Most of these words are acted upon immediately rather than compiled as they are
' part of the "compiler" in that they create the necessary structures
'

' dumb compiler for literals - improve later - just needs to optimize the number of bytes needed
LITCOMP  ' ( n -- ) compile the literal according to size
            byte    DUP,PUSH1,24,_SHR
            byte    _IF,@lco1-@LITC4
            ' Compile 4 bytes - 32bits
LITC4       byte    PUSH1,PUSH4,XCALL,xBCOMP
            byte     DUP,PUSH1,24,_SHR,XCALL,xBCOMP
            byte     DUP,PUSH1,16,_SHR,XCALL,xBCOMP
            byte     DUP,_8,_SHR,XCALL,xBCOMP
            byte     XCALL,xBCOMP,EXIT
lco1
            byte    DUP,PUSH1,16,_SHR
            byte    _IF,@lco2-@LITC3
            ' Compile 3 bytes - 24bits
LITC3       byte    PUSH1,PUSH3,XCALL,xBCOMP
            byte     DUP,PUSH1,16,_SHR,XCALL,xBCOMP
            byte     DUP,_8,_SHR,XCALL,xBCOMP
            byte     XCALL,xBCOMP,EXIT
lco2
            byte    DUP,_8,_SHR
            byte    _IF,@LITC1-@LITC2
            ' Compile 2 bytes - 16bits
LITC2       byte    PUSH1,PUSH2,XCALL,xBCOMP
            byte     DUP,_8,_SHR,XCALL,xBCOMP
            byte     XCALL,xBCOMP,EXIT
            ' Compile 1 byte - 8bits
LITC1       byte    PUSH1,PUSH1,XCALL,xBCOMP
            byte     XCALL,xBCOMP,EXIT

' MARK ( addr tag -- tag&addr ) Merge tag and addr by shifting tag into hi word
MARK        byte    _BYTE,$10,_SHL,_OR,EXIT
' UNMARK ( tag&addr -- addr tag )
UNMARK      byte    DUP,_WORD,$FF,$FF,_AND,SWAP,_BYTE,$10,_SHR,EXIT

' BEGIN as in BEGIN...AGAIN or BEGIN...UNTIL
_BEGIN_     byte    REG,codes,WFETCH,_BYTE,$BE,XCALL,xMARK                  ' generate branchs for BEGIN
bg01        byte    EXIT
' UNTIL ( flg -- )
_UNTIL_     byte    XCALL,xUNMARK
unt00       byte    _BYTE,$BE,EQ,_IF,@badthen-@unt01
unt01       byte    _BYTE,_UNTIL,XCALL,xBCOMP,JUMP,@calcback-@_REPEAT_  '
' AGAIN
_REPEAT_    byte    XCALL,xUNMARK
rp00        byte    _BYTE,$1F,EQ,_IF,@badrep-@rp02
rp02        byte    REG,codes,WFETCH,INC,INC,OVER,MINUS,SWAP,DEC,CSTORE' process branch of WHILE to after REPEAT
            byte    JUMP,@_AGAIN_-@badrep
badrep      byte    DROP2,JUMP,@badthen-@_AGAIN_
_AGAIN_     byte    XCALL,xUNMARK
ag00        byte    _BYTE,$BE,EQ,_IF,@badthen-@ag01                         '
ag01        byte    _BYTE,_AGAIN
            ' ( addr bc -- ) compile the bytecode and calculate the branch back
lpcalc      byte  XCALL,xBCOMP
calcback    byte    REG,codes,WFETCH,INC,SWAP,MINUS,XCALL,xBCOMP
            byte    EXIT

' IF as in IF...THEN or IF...ELSE...THEN
_WHILE_
_IF_        byte    _BYTE,_IF,XCALL,xBCOMP,_0,XCALL,xBCOMP
            byte    REG,codes,WFETCH,_BYTE,$1F,XCALL,xMARK
if01        byte    EXIT
' ELSE
_ELSE_      byte    XCALL,xUNMARK
el00        byte    _BYTE,$1F,EQ,_IF,@badthen-@el01                         ' does this match an IF?
el01        byte    _BYTE,JUMP,XCALL,xBCOMP,_0,XCALL,xBCOMP                 ' Compile a jump forward just like an IF
            byte    REG,codes,WFETCH,_BYTE,$1F,XCALL,xMARK                  ' mark the else to be processed on a THEN
el02        byte    SWAP,_BYTE,$1F,XCALL,xMARK                               ' get the IF addr and proceed as if it were a THEN
```

```
el03
' THEN
_THEN_      byte    XCALL,xUNMARK
th00        byte    _BYTE,$1F,EQ,_IF,@badthen-@RESFWD
RESFWD      byte    REG,codes,WFETCH,OVER,MINUS,SWAP,DEC,CSTORE,EXIT ' calculate branch and update IF's branch
badthen     byte    XCALL,xPRTSTR," Structure mismatch! ",0
            byte    DROP,EXIT

' " string"    Compile a literal string - no length restriction - any codes can be included except the delimiter "
_STR_       byte    _BYTE,XCALL,XCALL,xBCOMP,_BYTE,x_STR,XCALL,xBCOMP' compile bytecodes for string
            byte    JUMP,@COMPSTR-@st01
st01

' PRINT" HELLO WORLD"     Compile a literal print string - no length restriction - any codes can be included except the delimiter "
_PSTR_      byte    _BYTE,XCALL,XCALL,xBCOMP,_BYTE,xPRTSTR,XCALL,xBCOMP
COMPSTR
pslp        byte    XCALL,xWKEY,DUP,XCALL,xQEMIT                              ' echo string
            byte    DUP,_BYTE,$22,EQ,_IF,05,DROP,_0,XCALL,xBCOMP,EXIT
            byte    XCALL,xBCOMP,_AGAIN,@ps01-@pslp
ps01        ""
```

{ *** CASE STRUCTURE *** }

```
{ TACHYON CASE STRUCTURES
This implementation follows the C method to some degree.
Each CASE statement simply compares the supplied value with the SWITCH and executes an IF
To prevent the IF statement from falling through a BREAK is used (also acts as a THEN at CT)
The SWITCH can be tested directly and a manual CASE can be constructed with IF <code> BREAK

SWITCH ( switch -- )            \ Save switch value used in CASE structure
CASE ( val -- )          \ compare val with switch and perform an IF. Equivalent to SWITCH= IF
BREAK                    \ EXIT (return) but also completes IF structure at CT. Equivalent to EXIT THEN
\ extra functions to allow the switch to be manipulated etc
SWITCH@ ( -- switch )         \ Fetch last saved switch value
SWITCH= ( val -- flg )        \ Compare val with switch. Equivalent to SWITCH@ =
SWITCH>< ( from to -- flg )   \ Compare switch within range. Equivalent to SWITCH@ ROT ROT WITHIN

Usage:

pub CASEDEMO  ( val -- )
   SWITCH              \ use the key value as a switch in the case statement
   "A" CASE CR ." A is for APPLE " BREAK
   "H" CASE CR ." H is for HAPPY " BREAK
   "Z" CASE CR ." Z is for ZEBRA " BREAK
   $08 CASE 4 REG ~ BREAK
   \ Now accept 0 to 9 and build a number calculator style
   "0" "9" SWITCH>< IF SWITCH@ $30 - 4 REG @ #10 * + 4 REG ! ." *" BREAK
   \ On enter just display the accumulated number
   $0D CASE CR ." and our lucky number is " 4 REG @ .DEC BREAK
   \ show how we can test more than one value
   "Q" SWITCH= "X" SWITCH= OR IF CR ." So you're a quitter hey?" CR CONSOLE BREAK
   CR ." I don't know what " SWITCH@ EMIT ."  is"
    ;
pub DEMO
   BEGIN WKEY UPPER CASEDEMO AGAIN
 ;


            byte    $20,"BREAK",            hd+xc+im,       XCALL,xISEND
            byte    $20,"CASE",             hd+xc+im,       XCALL,xIS
            byte    $20,"SWITCH",           hd+xc,          XCALL,xSWITCH
            byte    $20,"SWITCH@",          hd+xc,          XCALL,xSWITCHFETCH
            byte    $20,"SWITCH=",          hd+xc,          XCALL,xISEQ
            byte    $20,"SWITCH><",         hd+xc,          XCALL,xISWITHIN
}


' SWITCH ( val -- )
_SWITCH  byte    REG,uswitch,STORE,EXIT
' SWITCH@ ( -- val )
SWITCHFETCH
            byte    REG,uswitch,FETCH,EXIT
' SWITCH= ( val -- flg )
ISEQ        byte    REG,uswitch,FETCH,EQ,EXIT
' CASE ( compare -- )
```

```
IS          byte    _BYTE,XCALL,XCALL,xBCOMP,_BYTE,xISEQ,XCALL,xBCOMP,XCALL,x_IF_,EXIT
' BREAK
ISEND       byte    _BYTE,EXIT,XCALL,xBCOMP,XCALL,xALLOCATED,XCALL,x_THEN_,EXIT


' SWITCH>< ( from to -- flg )..
ISWITHIN    byte    XCALL,xSWITCHFETCH,ROT,ROT,XCALL,xWITHIN,EXIT


{  Table vectoring -
index a table of vectors and jump to that vector
A table limit is supplied as well as a default vector

 Usage:
            <limit> VECTORS <vector if over>
            <vector0> <vector1> ..... <vectorx>)
Sample:
            4 LOOKUP BELL                                   \ an index of 4 or more will default to BELL
            INDEX0 INDEX1 INDEX2 INDEX3                     \ 0 to 3 will execute corresponding vectors

}
' LOOKUP
' VECTORS ( index range -- )
VECTORS     byte    OVER,GT,ZEQ,_IF,02,DROP,MINUS1                      ' limit index to range or -1 (.>0)
            byte    INC,_SHL1,RPOP,PLUS,CFETCHINC,SWAP,CFETCH           ' form index into vectors
EXECUTE     ' ( bytecode1 bytecode2 -- )                    ' 2 bytecodes
            byte    _WORD,(@bcexec+s)>>8,@bcexec+s
            byte    ROT,OVER,CSTORE,INC,CSTORE
bcexec      byte    EXIT,EXIT,EXIT

' XCALLS ( -- addr ) address of XCALLS vector table
_XCALLS     byte    _WORD,(@XCALLS+s)>>8,@XCALLS+s,EXIT



                                                { *** COMPILER *** }
' 12103 - adding a more general method for creating a call vector

' +CALL ( addr -- index opcode ) ' index = 0..$3FF
AddACALL    byte    XCALL,xXCALLS,_0
            byte    _2,PLUS,OVER,OVER,PLUS,WFETCH,ZEQ,_UNTIL,09         ' ( addr base index )
            byte    SWAP,OVER,PLUS,ROT,SWAP,WSTORE                      ' ( index )
            byte    _SHR1,DUP,_1,_AND,_BYTE,XCALL,SWAP,MINUS
            byte    OVER,_WORD,$02,00,_AND,ZEQ,ZEQ,_SHL1,PLUS
            byte    SWAP,_SHR1,toBYTE,SWAP
            byte    EXIT

' ( bytecode -- ) append this bytecode to next free code location + append EXIT (without counting)
BCOMP
            byte    REG,codes,WFETCH,CSTORE,_1,REG,codes,WPLUSST
            byte    _BYTE,EXIT,REG,codes,WFETCH,CSTORE
            byte    EXIT

COMPILE     byte    XCALL,xGETWORD,DEC,XCALL,xSEARCH,EXIT   ' get next word, search for it in dictionary
            byte  XCALL,xNFATICK,DEC

' ( atradr -- ) compile bytecodes according to attribute - 0 = one bytecode ; 2 = 2 bytecodes
BCOMPILE
            byte    CFETCHINC,_3,_AND
            byte    DUP,ZEQ,_IF,05,DROP,CFETCH,XCALL,xBCOMP,EXIT
            byte    DUP,_2,EQ,_IF,08,DROP,CFETCHINC,XCALL,xBCOMP,CFETCH,XCALL,xBCOMP,EXIT
            byte    DROP2,EXIT

' GRAB ( -- ) \ IMMEDIATE --- executes preceding code to make it available for any immediate words following
GRAB
            byte    PUSH1,EXIT,XCALL,xBCOMP                             ' append an EXIT
            byte    XCALL,xHERE,DUP,REG,codes,WSTORE,ACALL              ' execute and release preceding code in text line
            byte    EXIT

' assign a new value for the constant
' 78 TO myconstant
ATO
            byte    XCALL,xGRAB,XCALL,xTICK,INC,STORE,EXIT

' ( -- atradr ) --- point to the attribute byte in the header of the latest name
ATATR       byte    REG,names,WFETCH,XCALL,xNFACFA,DEC,EXIT

' Set attribute of the latest word to PRIVATE -- used by RECLAIM (EXTEND.fth) to cull all unwanted headers.
PRIVATE     byte    XCALL,xCOLON,_BYTE,pr,XCALL,xATATR,SET,EXIT

'  CREATEWORD - create a name in the dictionary using the next word encountered
```

```
CREATEWORD
            byte    XCALL,xGETWORD                                  ' ( str ) read the next word
' (CREATE) ( str -- )
CREATESTR
            byte    REG,names,WFETCH,REG,names+2,WSTORE             ' backup names ptr (used to change fixed fields easily)
            byte    REG,flags,CFETCH,_BYTE,prset,_AND               ' get attribute
            byte    _BYTE,hd+xc,_OR                                 ' blend in private bit
            byte    XCALL,xPUTCHARPL                                  ' add attribute byte
            byte    REG,codes,WFETCH                                  ' Create a vector
            byte    XCALL,xAddACALL,XCALL,xPUTCHARPL,XCALL,xPUTCHARPL' write opcode + index bytecode sequence

            byte    DUP,DEC,CFETCH   'XCALL,xSTRLEN                   ' ( str cnt )
            byte    DUP,NEGATE,REG,names,WPLUSST                    ' ( str cnt ) update names ptr by backwards count
            byte    REG,names,WFETCH,SWAP,XOP,pCMOVE                ' copy it across
            byte    REG,names,WFETCH,DUP,XCALL,xSTRLEN              ' ( names cnt )
            byte    MINUS1,REG,names,WPLUSST                        ' make room for the count
            byte    SWAP,DEC,CSTORE                                   ' and set the count
            ' check for dictionary full
            byte    REG,names,WFETCH,REG,here,WFETCH,_BYTE,$40,PLUS,XCALL,xLT
            byte    _IF,@crw05-@crw04
crw04       byte    XCALL,xPRTSTR,"  Dictionary full! ",0
            byte    XCALL,xERROR
crw05       byte    EXIT

' CREATE <name> - Create a name in the dictionary and also a VARIABLE code entry - or execute call at create
CREATE      byte    REG,createvec,WFETCH,QDUP,_IF,01,AJMP           ' execute extended or user CREATE?
            byte    XCALL,xCREATEWORD                                 '
            byte    _BYTE,VARB,XCALL,xBCOMP,_0                      ' set default bytecode as a VARIABLE

' ALLOT ( bytes -- )
ALLOT       byte    REG,codes,WPLUSST
' lock in compiled code so far - do not release but set new "here" to the end of these codes
ALLOCATED
            byte    REG,codes,WFETCH,REG,here,WSTORE
            byte    EXIT

' C, ( n -- ) IMMEDIATE --- compile a byte into code and allocate
CCOMP
            byte    XCALL,xGRAB
cc01        byte    XCALL,xBCOMP,XCALL,xALLOCATED,EXIT
' W, ( n -- )
WCOMP
            byte    XCALL,xGRAB
wc01        byte    DUP,XCALL,xBCOMP,_8,_SHR,XCALL,xBCOMP,XCALL,xALLOCATED,EXIT

' , ( n -- ) Compile a long literal
LCOMP
            byte    XCALL,xGRAB
            byte    _4,FOR,DUP,XCALL,xBCOMP,_8,_SHR,forNEXT
            byte    DROP,XCALL,xALLOCATED,EXIT

' Create a new entry in the dictionary and also in the XCALLS table but also prevent any execution of code
' at an <enter> which would otherwise normally occur.
' unsmudge any previous name in case this is a fall-through.
' : <name>
COLON       byte    XCALL,xUNSMUDGE                                 ' unsmudge any previous definition (fall-through)
            byte    XCALL,xCREATE                                   ' this forms an XCALL,index to this new definition
_COLON      byte    _BYTE,sm,XCALL,xATATR,SET                       ' smudge it so it can't be referenced yet
            byte  MINUS1,XCALL,xALLOT                               ' write over VAR instruction
            byte    _BYTE,defining,REG,flags,SET,EXIT               ' flag that we have entered a definition

PUBCOLON        byte        XCALL,xCOLON,_BYTE,prset,XCALL,xATATR,CLR,EXIT

' Update "here" pointer to point to current free position which "codes" pointer is now at
' Also unsmudge the headers tag
' ;
ENDCOLON        byte        _BYTE,EXIT,XCALL,xBCOMP                 ' compile an EXIT
            byte    _BYTE,defining,REG,flags,CLR,XCALL,xALLOCATED   ' end definition and lock allocated bytes
UNSMUDGE
            byte    _BYTE,sm,XCALL,xATATR,CLR,EXIT                  ' clear the smudge bit


' [COMPILE]
COMPILESbyte    _BYTE,comp,REG,flags,SET,EXIT
```

{ *** CONSOLE INPUT HANDLERS *** }

```
{
Replaced traditional parse function with realtime stream parsing
Each word is acted upon when a delimiter is encountered and this also allows for
```

```
interactive error checking and even autocompletion.
}

' SCRUB --- scrub out any temporary compiled code, restore the code pointers etc.
SCRUB     byte    XCALL,xHERE,REG,codes,WSTORE
          byte    _0,REG,wordcnt,CSTORE,_0,REG,wordbuf,CSTORE
          byte    _BYTE,$0D,REG,delim+1,CSTORE                                    'restore end-of-line delimiter to a CR
          byte    _BYTE,$0D,XCALL,xEMIT,_BYTE,$40,FOR,_BYTE,"-",XCALL,xEMIT,forNEXT    'horizontal line
          byte    XCALL,xCR,EXIT

' ( ch -- ) write a character into the next free position in the word buffer
PUTCHAR byte    REG,wordcnt,DUP,CFETCH,SWAP,INC,PLUS,CSTORE,EXIT
PUTCHARPL       byte        XCALL,xPUTCHAR,REG,wordcnt,DUP,CFETCH,INC
          byte    _BYTE,wordsz,UDIVMOD,DROP,SWAP,CSTORE,EXIT

' As characters are accepted from the input stream, checks need to be made for delimiters,
' editing commands etc. 123us/CHAR, 184us/CTRL
doCHAR    ' ( char -- flg ) Process char into wordbuf and flag true if all done
          byte    DUP,ZEXIT                                                 ' NULL - ignore
          '
          byte    DUP,REG,lasttwo,DUP,CFETCH,OVER,INC,CSTORE,CSTORE    ' keep a track of this and the last character
          byte    DUP,REG,delim+1,CSTORE                              ' delimiter is always last character
          '''
          byte    _BYTE,$7F,OVER,EQ,_IF,02,DROP,_8                    ' subs BS for DEL
          byte    DUP,BL,XCALL,xLT,_IF,@ischar-@ctrls                 ' only check for control characters
'
' PROCESS CONTROL CHARACTERS
'
ctrls
'         byte    _BYTE,$80,OVER,_AND,_IF,04,_1,REG,flags+1,SET          ' IAC or binary
          byte    _BYTE,$0A,OVER,EQ,_IF,03,DROP,_FALSE,EXIT           ' LF - discard
          byte    _BYTE,$18,OVER,EQ,_IF,03,DROP,_TRUE,EXIT            ' ^X reeXecute previous compiled line
          byte    _1,REG,flags+1,XCALL,xSETQ,ZEQ,_IF,@ignore2-@ignore1
ignore1
          byte    _3,OVER,EQ,_IF,02,XCALL,xREBOOT                     ' ^C RESET
          byte    _4,OVER,EQ,_IF,05,DROP,XCALL,xDEBUG,_FALSE,EXIT     ' ^D DEBUG
          byte    _2,OVER,EQ,_IF,09,DROP,_0,_WORD,$80,00,XCALL,xDUMP,_FALSE,EXIT          ' ^B Block dump
          byte    _BYTE,$1A,OVER,EQ,REG,lasttwo+1,CFETCH,_BYTE,$1A,EQ,_AND
          byte    _IF,07,DROP,XCALL,xCOLDST,XCALL,xSCRUB,_FALSE,EXIT  ' ^Z^Z cold start
ignore2
          byte    _BYTE,$1B,OVER,EQ,_IF,05,DROP,XCALL,xSCRUB,_TRUE,EXIT' ESC will cancel line
          byte    _BYTE,$09,OVER,EQ,_IF,03,XCALL,xEMIT,BL             ' TAB - substitute with a space
          byte    _BYTE,$1C,OVER,EQ,_IF,04,DROP,XCALL,xCR,BL            ' ^| - multi-line interactive
          byte    _BYTE,$0D,OVER,EQ,_IF,03,DROP,_TRUE,EXIT              ' CR - Return & indicate completion
          '
          byte    _8,OVER,EQ,_IF,@ischar-@bksp1                       ' BKSP - null out last char
bksp1     byte    REG,wordcnt,CFETCH,_IF,@bksp3-@bksp2                ' don't backspace on empty word
bksp2     byte    XCALL,xEMIT,XCALL,xSPACE,_8,XCALL,xEMIT             ' backspace and clear
          byte    MINUS1,REG,wordcnt,CPLUSST,_0,XCALL,xPUTCHAR        ' null previous char
          byte    _FALSE,EXIT
          '                           '
bksp3     byte    _BYTE,7,XCALL,xEMIT,DROP,_FALSE,EXIT                ' can't backspace anymore, bell
          '
ischar
          byte    _BYTE,echo,REG,flags,XCALL,xSETQ,_IF,03             ' don't echo if we don't want it
          byte    DUP,XCALL,xEMIT
          byte    REG,delim,CFETCH,OVER,EQ                            ' delimiter? (always accept a blank)
          byte    OVER,BL,EQ,_OR,_IF,05,DROP,REG,wordcnt,CFETCH,EXIT  ' true if trailing delimiter - all done (flg=cnt)
          '
          ' otherwise build text in wordbuf - null terminated with a preceding count .....
          byte    XCALL,xPUTCHARPL                                      ' put a character into the word buffer
          byte    _FALSE,EXIT

' Build a delimited word and return immediately upon a valid delimiter
GETWORD         ' ( -- str ) Build a text word from character input into wordbuf for wordcnt
          byte    REG,wordcnt,PUSH1,wordsz,_0,XCALL,xFILL                 'Erase the word buffer & preceding count
gwlp      byte    XCALL,xWKEY                                             ' get another character
          byte    XCALL,xdoCHAR                                           ' process
          byte    _UNTIL,@gw1-@gwlp                                        'continue building the next word
gw1       byte    REG,wordbuf,EXIT
```

{ *** DICTIONARY SEARCH *** }

```
{ DICTIONARY SEARCH

Example of last entry in dictionary "COLD"  04 43 4F 4C 44 82  BF  A4  00 00
                                 cnt C  O  L  D  atr bc1 bc2 <end>

"
```

```
" Compare a null-terminated source string with a dictionary string which is 8th bit terminated.
" This will always force a mismatch after which one is checked for a null while the other is checked
" for the 8th bit and if verified then a match has been found.
" The dict pointer is advanced to point to the end of the dict string on the 8th bit termination which
" is the attribute byte as in: byte "CMPSTR",$80,CMPSTR
"

}
{ search timing results
DUP         54us
RESET       1ms
0           144us
BL          288us
KOLD        3.45ms
TAB         9ms
EXTEND.fth          7.87ms      9.85ms
12345       600us
}
SEARCH      ' ( cstr -- nfaptr )                                ' cstr points to the count+string+null
{ extra method fpr searching user dictionary first
            byte    REG,unames,WFETCH,QDUP,IF,@sr02-@sr01                       ' user dictionary?
sr01
            byte    OVER,SWAP,XCALL,xFINDSTR                                    ' search user words overrides all other searches
            byte    QDUP,_IF,02,NIP,EXIT                        ' found it - return now with result
}
sr02
            byte    REG,findvec,WFETCH,QDUP,_IF,01,AJMP                         ' use alternative method if enabled (hash search)
            byte    DUP,REG,corenames,WFETCH,XCALL,xFINDSTR                     ' search core words first to improve compilation speed
            byte    QDUP,_IF,02,NIP,EXIT                        ' found it - return now with result
'
            byte    DUP,REG,names,WFETCH,XCALL,xFINDSTR                         ' search extended dictionary
            byte    QDUP,_IF,02,NIP,EXIT                        ' return with positive result
'
'           byte    REG,findvec,WFETCH,QDUP,_IF,01,AJMP                         ' user specified search
            byte    DROP,_FALSE,EXIT                                            ' not found in dictionary



{
' CMPSTR ( src dict 0 -- src dict flg ) Compare strings at these two addresses
' this is the RUNMOD version which gets loaded during a block compile typically when the app is not running
                    org     _RUNMOD
rCMPSTR

' CMPSTR ( src dict flg-- src dict flg ) Compare strings at these two addresses
            if_never rdbyte $1ff,#$1ff  ' dummy op - signature to detect this module is loaded $00C3_FFFF
                    mov     R2,tos+1 wc                 ' R2=dict : force nc on entry, s[31] == 0 (doNEXT clears c)
                    mov     X,tos+2                     ' X = source
cmpstrlp                    rdbyte  R0,X                ' read in a character from the source
                    add     X,#1                        ' hub has to wait anyway so get ready for next source byte
                if_c add    R2,#1                       ' updates the copy of the dictionary pointer
                    rdbyte  R1,R2                       ' read in from the dictionary
                    cmp     R1,R0 wz                    ' are they the same?
                if_z jmpret par,#cmpstrlp wc,nr         ' keep at it, set carry to enable dict+ (for local copy)
nomatch                     cmp     R0,#1 wc            ' was the src null terminated? (C means we have matched the string)
                if_c test   R1,#$80 wc                  ' set c flag if dict 8th bit set (C = cross-matched)
                    jmp     #SETZ                       ' Change our flag to -1 if C set (matched)

CMPSTRMOD       byte        _WORD,(@rCMPSTR+s)>>8,@rCMPSTR+s,XCALL,xLOADMOD,EXIT
}
{ Testing fast CMPSTR and slow CMPSTR
4000 LAP SEARCH LAP .LAP  6.78ms ok
0 1DD COG!  ok
4000 LAP SEARCH LAP .LAP  12.65ms ok

}
{
' CMPSTR ( src dict -- src dict flg ) Compare strings at these two addresses
' This version checks first if the RUNMOD version is loaded and runs that instead, otherwise the slower bytecode version
CMPSTR
            byte    _WORD,pRUNMOD>>8,pRUNMOD,XOP,pCOGFETCH
            byte    _LONG,$00,$C3,$FF,$FF,EQ,_IF,04,_0,XOP,pRUNMOD,EXIT         ' pCMPSTR loaded so run it ( src dict flg )
            ' ( src dict -- src dict flg )
            byte    OVER,OVER                                                   ' save src dict to restore later ( src dict src dict )
cslp        byte    OVER,CFETCH,OVER,CFETCH,EQ,_IF,06,INC,SWAP,INC,SWAP,_AGAIN,@cs01-@cslp
cs01
            byte    OVER,CFETCH,ZEQ,OVER,CFETCH,_BYTE,$7F,GT,_AND
            ' ( src dict src+ dict+ flg )
            byte    NIP,NIP,EXIT
}

' ( cstr dict -- nfaptr | false ) Try to find the counted string in the dictionary(s) using CMPSTR  (ignore smudged entries)
FINDSTR
fstlp               'byte       XCALL,xCMPSTR
            byte    _0,XOP,pCMPSTR
            byte    _IF,@nxtword-@fst1                                      ' found it  ( src dict )
fst1        byte    DUP,XCALL,xNFACFA,DEC,CFETCH
            byte    _BYTE,sm,_AND,ZEQ,_IF,@nxtword-@fst0
fst0        byte    NIP,EXIT                                                ' ( nfaptr ) found
            '
```

```
                ' Skip the attribute byte and codes and test for end of dictionary (entry = 00)
nxtword         ' ( src dict ) advance past atr+codes to try next.  (atr(1),bytecode)
                byte    CFETCHINC,PLUS,_3,PLUS                                  ' jump over CFA to next NFA
                                                                                ' ( src dict ) dict points to bc1
                byte    DUP,CFETCH,ZEQ,_UNTIL,@fst2-@fstlp
                ' end of dictionary reached
fst2            byte    DROP2,_FALSE,EXIT



' The CFA is the address of the 2 bytecodes stored in the header that are executed or compiled
' Typically these bytecodes will be in the form of "XCALL,xWord"
' or in the case of COG words such as DUP "DUP,EXIT" where only DUP is compiled
' NFA>CFA ( nfa -- cfa )  BEGIN C@++ $7F > UNTIL ;
NFACFA          byte    CFETCHINC,_BYTE,$7F,GT,_UNTIL,06,EXIT

' : >PFA  ( cfa -- pfa )
PFA             byte    DUP,DEC,CFETCH,_BYTE,$80,EQ,_IF,02,CFETCH,EXIT  ' if atr = COG BYTECODE - just return with it
                ' but this could be a two bytecode sequence rather than a call
                byte    DUP,CFETCH,_BYTE,XOP,EQ,_IF,07,CFETCHINC,_8,_SHL,SWAP,CFETCH,PLUS,EXIT
' \ MJB          byte    DUP,CFETCH,_BYTE,XOP,EQ,_IF,05,1+,CFETCH,_16,PLUS,EXIT  ' saves 2 bytes and a hubop and works whether XOP is $01 or not (as it is now $0B)

                byte    XCALL,xTOVEC
                byte    WFETCH,EXIT

' >VEC ( cfa -- vecptr )
TOVEC           byte    DUP,INC,CFETCH,_SHL1,_SHL1,XCALL,xXCALLS,PLUS ' ( cfa xcallptr )
                byte    SWAP,CFETCH,_BYTE,VCALL,MINUS                           ' calculate which vector XYZ or V we are using
                byte    _3,_XOR                                                 ' ( xcallptr mod )
                ' ( ptr indexh ) 0 = XCALL, 1 = YCALL, 2 = ZCALL, 3 = VCALL
                byte    SWAP,OVER,_1,_AND,_SHL1,PLUS                            ' point to high or low vector word
                byte    SWAP,_2,_AND,_IF,04,_WORD,$04,00,PLUS
                byte    EXIT


' NFA'
NFATICK         byte    XCALL,xGETWORD,DEC,XCALL,xSEARCH,EXIT
_NFATICK        byte    XCALL,xNFATICK,XCALL,xLITCOMP,EXIT

' ' <name>  ( -- pfa ) Find the address of the following word - zero if not found or it's PFA (bytecodes do not have a CFA)
TICK            byte    XCALL,xNFATICK,DUP,ZEXIT,XCALL,xNFACFA,XCALL,xPFA,EXIT

ATICK           byte    XCALL,xTICK
                byte    XCALL,xLITCOMP,EXIT


' removed SETPOLL from here
' SETPOLL       byte    XCALL,xTICK,REG,keypoll,WSTORE,EXIT
{
WORDS           byte    REG,names,WFETCH
wdlp
                byte    DUP,CFETCH,ZEQ,DUP,_IF,06,SWAP,INC,DUP,CFETCH,ZEQ,ROT,_AND,_IF,@wd03-@wd02
wd02            byte    DROP,XCALL,xCR,EXIT

wd03            byte    XCALL,xCR,DUP,XCALL,xPRTWORD,XCALL,xPRTSTR,": ",0
                byte    INC,DUP,XCALL,xNFACFA,DEC,DUP,_3
                byte    ADO,I,CFETCH,XCALL,xPRTBYTE,XCALL,xSPACE,LOOP
                byte    _3,PLUS,SWAP,XCALL,xPSTR,XCALL,xSPACE,_AGAIN,@wd04-@wdlp
wd04
}

' AUTORUN <name>     - Setup Tachyon to AUTORUN the word on reset - an invalid or no name will disable AUTORUN
' <name> must be a valid user word so it must be an XCALL
AUTORUN         byte    XCALL,xTICK
' AUTO! ( addr -- ) Set the AUTORUN vector
AUTOST
setauto         byte    REG,autovec,WSTORE,EXIT



' FREE ( -- free ) Read free memory from Spin header and round up to a 64 byte page (to suit EEPROM)
'FREE           byte    _BYTE,10,WFETCH,_BYTE,$80,PLUS,_BYTE,$3F,_ANDN,EXIT
FREE            byte    _WORD,(@last+s)>>8,@last+s,_BYTE,$80,PLUS,_BYTE,$3F,_ANDN,EXIT

' correct the count byte in each entry in the dictionary as the precompiled version leaves a dummy count (too mind-numbing)
FIXDICT         byte    _WORD,(@dictionary+s)>>8,@dictionary+s
fdlp            byte    DUP,INC,XCALL,xSTRLEN,OVER,CSTORE                       ' calc length and set count byte
                byte    DUP,CFETCH,_4,PLUS,PLUS,DUP,CFETCH,ZEQ,_UNTIL,@fd01-@fdlp
fd01            byte    DROP,EXIT

' COLD           Force factory defaults
COLDST          byte    XCALL,xPRTSTR," Cold start - no user code - setting defaults ",$0D,$0A,0
                byte    XCALL,xFREE,DUP,REG,here,WSTORE,REG,here-2,WSTORE       ' free memory
                byte    _WORD,(@rxbuffers+s)>>8,@rxbuffers+s,REG,rxptr,WSTORE   ' setup saved receive buffer address
                byte    _WORD,(@dictionary+s)>>8,@dictionary+s
                byte    DUP,REG,corenames,WSTORE,DEC
                byte    DUP,REG,names,WSTORE,REG,names-2,WSTORE                 ' Reset dictionary pointer
                byte    _0,REG,autovec,WSTORE,_0,REG,keypoll,WSTORE             ' disable autorun and ext keypoll
                byte    XCALL,xFIXDICT

                byte    _BYTE,xLAST,DUP,_SHL1,_SHL1,XCALL,xXCALLS,PLUS          ' find first free XCALL memory location
                byte    _WORD,$02,00,ROT,MINUS,_SHL1,_SHL1,_0,XCALL,xFILL       ' clear all user XCALLs
                byte    XCALL,xXCALLS,INC,INC,_WORD,$08,00                      'clear YCALLs (interleaved with XCALLS)
```

```
                byte    ADO,_0,I,WSTORE,_4,PLOOP
                byte    REG,tasks,_BYTE,tasksz*8,_0,XCALL,xFILL                 ' initialize 8 task words

                byte    _WORD,$A5,$5A,REG,cold,WSTORE
                byte    EXIT

' Discard the current line
DISCARD
dslp            byte    XCALL,xKEY,ZEQ,_UNTIL,@ds01-@dslp
ds01            byte    _BYTE,100,XCALL,xms,XCALL,xKEY,ZEQ,_UNTIL,@ds02-@dslp
ds02            byte    EXIT

' TASK ( cog -- addr ) Return with address of task control register in "tasks"
TASK            byte    _3,_SHL,REG,tasks,PLUS,EXIT
{
'RUN ( pfa cog -- )
RUN             byte    XCALL,xTASK,WSTORE,EXIT
}
                org                     ' align the label - needs to be passed in 14-bit PAR register
' idle loop for other Tachyon cogs
IDLE_reset      byte    pInitRP                                                 ' cog reset entry
IDLE            byte    XOP,pInitRP
id02            byte    XCALL,xInitStack
                byte    _1,XOP,_COGID,XCALL,xTASK,INC,INC,CSTORE                ' task+2 = 1 to indicate Tachyon running
idlp            byte  _8,XCALL,xms                                      ' do nothing for a bit - saves power
                byte    XOP,_COGID,XCALL,xTASK,WFETCH                           ' fetch cog's task variable
                byte    QDUP,_UNTIL,@id00-@idlp                                 ' until it is non-zero
id00            byte    DUP,_8,_SHR,_IF,01,ACALL                                ' Execute but ignore if task address is only 8-bits
                byte    _0,XOP,_COGID,XCALL,xTASK,WSTORE                        ' clear run address only if it has returned back to idle
                byte    _AGAIN,@id01-@IDLE
id01
```

{ *** MAIN TERMINAL CONSOLE *** }

```
                org     ' align the label for RESET
TERMINAL_reset
                byte    pInitRP                                                 ' Only reset from a cog enters here
TERMINAL
                byte    XOP,pInitRP                                             ' normal entry
SETPLL          byte    _WORD,(@_setpll+s)>>8,@_setpll+s                        ' autoset crystal operation
                byte    XCALL,xLOADMOD,XOP,pRUNMOD
                byte    _BYTE,txd,MASK,OUTSET                                   ' be a friend and make the transmit an output
                byte    XCALL,xInitStack                                        ' Init the internal stack and setup external stack space
                byte    _WORD,extstk>>8,extstk,XCALL,xSPSTORE
                byte    _WORD,00,50,XCALL,xms                                   ' a little startup delay (also wait for serial cog)
                byte    _WORD,rxsize>>8,rxsize,REG,rxsz,WSTORE                  ' Set the rx buffer size
                byte    _BYTE,echo,REG,flags,CSTORE                             ' echo on
                byte    BL,REG,delim,CSTORE,XCALL,xHEX                          ' default delimiter is a space character
                byte    _BYTE,$0D,XCALL,xEMIT,XCALL,xPRTVER                     ' Show VERSION with optional CLS (default CR)
'               byte    XCALL,xCMPSTRMOD                                        ' use fast string compare module
                byte    REG,cold,WFETCH,_WORD,$A5,$5A,EQ,ZEQ                    ' performing a check for a saved session
                byte    _IF,@warmst-@tm01                                       ' or not
tm01            byte    XCALL,xCOLDST                                           ' defaults

warmst
                byte    REG,lastkey,CFETCH,_1,EQ,_NOT,_IF,@termnew-@chkauto     ' ^A abort autostart with ^A
chkauto         byte    REG,autovec,WFETCH,QDUP,_IF,@termnew-@runauto           ' check for an AUTORUN
runauto         byte    ACALL                                                   ' and execute it
CONSOLE
termnew         byte    XOP,pInitRP                                             ' init return stack in case (limited)
                byte    XCALL,xSCRUB
                ' Main console line loop - get a new line (word by word)
termcr          byte    REG,prompt,WFETCH,QDUP,_IF,01,ACALL                     ' execute user prompt code
                byte    XCALL,xHERE,REG,codes,WSTORE                            ' reset temporary code compilation pointer

                '
                ' Main console loop - read a word and process
termlp
                byte    XCALL,xGETWORD                                          ' Read a word from input stream etc

                byte    CFETCH,ZEQ,_IF,@trm1-@trm2                              ' ignore empty string
trm2            byte    REG,delim+1,CFETCH,_BYTE,$18,EQ,_NOT,_IF,@execinp-@trm3 ' ^X then repeat last line
trm3            byte    REG,delim+1,CFETCH,_BYTE,$0D,EQ,_NOT,_IF,@chkeol-@trm1  ' Otherwise process ENTER
                '
trm1            ' Preprocess prefixed numbers #$%
                byte    REG,wordbuf,CFETCH,_BYTE,"#",_BYTE,"%",XCALL,xWITHIN    ' Numeric prefixes?
                byte    REG,wordbuf-1,CFETCH,_2,GT,_AND                         ' and more than 2 characters? (inc term)
                byte    REG,wordbuf-1,DUP,CFETCH,PLUS,CFETCH                    ' and last char is a digit or hex digit?
                byte    DUP,_BYTE,"0",_BYTE,"9",XCALL,xWITHIN                   ' decimal digit?
                byte    SWAP,_BYTE,"A",_BYTE,"F",XCALL,xWITHIN,_OR,_AND ' hex digit?
                byte    ZEQ,_IF,@tryanum-@trm4                                  ' good, process this as a number now @tryanum

                ' Search the dicitonary for a match (as a counted string)
trm4            byte    REG,wordbuf,DEC,XCALL,xSEARCH                           ' try and find that word in the dictionary(s)
                byte    QDUP,_IF,@_notfound-@foundword                          ' found it

foundword                                                                       ' found the word in the dictionary - compile or execute?
                byte    XCALL,xNFACFA,DEC                                       ' point to attribute byte
                byte    PUSH1,im,OVER,XCALL,xSETQ                               ' is the immediate bit set?
```

```
                byte    _BYTE,comp,REG,flags,XCALL,xSETQ,ZEQ,_AND                ' and comp flag off (not forced to compile)
                byte    _IF,@compword-@immed                                     '
immed           byte    INC,CFETCHINC,SWAP,CFETCH,XCALL,xEXECUTE                 ' Fetch and EXECUTE code immediately
                byte    _ELSE,@chkeol-@compword
compword        byte    XCALL,xBCOMPILE                                          ' or else COMPILE the bytecode(s) for ths word
                byte    _BYTE,comp,REG,flags,CLR                                 ' reset any forced compile mode
                ' END OF LINE CHECK
chkeol          byte    REG,delim+1,CFETCH,_BYTE,$0D,EQ                          ' Was this the end of line?
                byte    DUP,_IF,@eol01-@eol00
eol00           byte    REG,accept,WFETCH,ZEQ,_IF,02,XCALL,xSPACE                ' Yes, put a space between any user input and response
                byte    _BYTE,linenums,REG,flags,XCALL,xSETQ,_IF,@eol01-@prtline
prtline         byte    _BYTE,$0D,XCALL,xEMIT                                    ' List line number if enabled
                byte    REG,linenum,WFETCH,XCALL,xPRTDEC,XCALL,xSPACE
                byte    _1,REG,linenum,WPLUSST
eol01           byte    DUP,_BYTE,defining,REG,flags,XCALL,xSETQ,_AND             ' and are we in a definition or interactive?
                byte    _IF,02,XCALL,xCR                                         ' If not interactive then CRLF (no other response)
                byte    _BYTE,defining,REG,flags,XCALL,xSETQ,ZEQ,_AND            ' do not execute if still defining
                byte    _UNTIL,@execs-@termlp                                    ' wait until CR to execute compiled codes
                ' EXECUTE CODE from user input
execs           byte    PUSH1,EXIT,XCALL,xBCOMP                                  ' done - append an EXIT (minimum action on empty lines)
execinp         byte    XCALL,xHERE,ACALL                                        ' execute from beginning
                byte    REG,accept,WFETCH                                        ' if accept vector is <>0
                byte    QDUP,_IF,03,ACALL                                        ' then execute it
                byte    _ELSE,02,XCALL,xOK                                       ' else echo the "ok"
                byte    _AGAIN,@_notfound-@termcr

_notfound       ' NOT FOUND YET - before trying to convert to a number check encoding for ASCII literals (^ and ")
                ' Attempt to process this word as a number
tryanum         byte    REG,wordbuf,XCALL,xNUMBER,_IF,@unknown-@compnum
compnum         byte    XCALL,xLITCOMP
                byte    _AGAIN,@unknown-@chkeol                                  ' is it a number? ( value digits )

' Unknown word or number - try converting case
UNKNOWN
                byte    REG,wordbuf,CFETCH,_BYTE,$60,GT                          ' failed to find it - basic check of case - first char lower?
                byte    _IF,06,REG,wordbuf,XCALL,xTOUPPER,_AGAIN,@un03-@trm4     ' auto convert to uppercase if not found
un03            byte    REG,unum,WFETCH,QDUP,_IF,03,ACALL,_AGAIN,@un01-@chkeol   ' UNKNOWN - try unum vector if set
un01            byte    XCALL,xNOTFOUND
un02            byte    _AGAIN,@NOTFOUND-@termlp

' Failed all searches and conversions
NOTFOUND
                byte    XCALL,xPRTSTR," --> ",0
                byte    REG,wordbuf,XCALL,xPSTR                                  ' Spit out offending word
                byte    XCALL,xPRTSTR," <-- not found ",7,$0D,$0A,0              ' --> xxx <--- NOT FOUND
ERROR           byte    _1,REG,errors,WPLUSST                                    ' count errors
'               byte    _BYTE,7,XCALL,xEMIT
                byte    XCALL,xDISCARD,EXIT

' TACHYON - used to verify that source code is intended for Tachyon and also to reset load stats
_TACHYON        byte    XCALL,xPRTVER
                byte    _0,REG,keypoll,WSTORE                                    ' disable background keypoll during load
                byte    _0,REG,errors,WSTORE
'               byte    XCALL,xCMPSTRMOD                                         ' use fast string compare module
                byte    XCALL,xHERE,REG,here-2,WSTORE                            ' remember code position
                byte    _0,REG,linenum,WSTORE,_BYTE,linenums,REG,flags,SET       ' reset line# and set linenum mode
                byte    REG,names,WFETCH,REG,names-2,WSTORE                      ' backup dictionary pointer
                byte    LAP,EXIT                                                 ' time the load

' VER ( -- verptr )
GETVER          byte    _WORD,(@version+s)>>8,@version+s,EXIT

' .VER
PRTVER          byte    XCALL,xPRTSTR,$0D,$0A
                byte      "  Propeller .:.:--TACHYON--:.:. Forth V",0
                byte    XCALL,xGETVER,DUP,FETCH,XCALL,xPRTDEC
                byte    _BYTE,".",XCALL,xEMIT,_4,PLUS,WFETCH,XCALL,xPRTDEC
                byte    XCALL,xCR,EXIT


{
```

****

********** METACOMPILED CODE & HEADER MEMORY **********

****

```
}
```

```
last
TRIM            byte    0[$5300-$]              ' trim this manually to optimize top of memory (buffers @7500 etc)
```

{ *** DICTIONARY in RAM and EEPROM *** }

{

The Forth dictionary is loaded into high RAM and is not used at runtime normally unless the console is used to "talk" to Forth.

Search methods:

Structure:
1 - Count byte - This speeds up searching the dictionary both in comparing and in jumping to the next string
2- Name string
3- Attribute byte (8th bit set also terminates name string)
4- 1st bytecode, 2nd bytecode

Dictionary entries do not need a link field as they are bunched together one after another and it is very easy to find the next entry by scanning forwards and looking for the attribute byte which will have the msb set then jumping 3 bytes. A name field that begins with a null indicates end of dictionary (or link to another), null null is the end.
}

CON

```
' Dictionary header attribute flags
hd          = |<7           'indicates this is a an attribute (delimits the start of a null terminated name)
sm          = |<6           'smudge bit - set to deactivate word during definition
im          = |<5           'lexicon immediate bit

pr          = |<3           'private (can be removed from the dictionary)
' code attributes    00 = single bytecode, 02 = XCALL bytecode (2 bytes), 03 = WCALL bytecode (3 bytes)
sq          = |<2           'indicates the bytecode is a sequence of two PASM instructions (as opposed to a vectored call)
xc          = |<1           'XCALL bytecode
ac          = xc+|<0        'WCALL - 2 byte address - interpret header CFA as an absolute address
```

DAT

```
{ This is an 8th bit terminated string using the attribute byte so it saves one byte per entry plus it simplfies the string compare function. Searching still proceeds from lowe
memory to higher memory
}
```

{ *** DICTIONARY *** }

dictionary

' The count field is left blank but filled in at runtime so that these do not need to be calculated when defining
'

```
'       CNT,NAME            ATR         CODES

        byte    $20,"DUP",          hd,         DUP,EXIT
        byte    $20,"OVER",         hd,         OVER,EXIT
        byte    $20,"DROP",         hd,         DROP,EXIT
        byte    $20,"2DROP",        hd,         DROP2,EXIT
        byte    $20,"SWAP",         hd,         SWAP,EXIT
        byte    $20,"ROT",          hd,         ROT,EXIT
        byte    $20,"NIP",          hd,         NIP,EXIT
        byte    $20,"BOUNDS",       hd,         BOUNDS,EXIT
'       byte    $20,"REPS",         hd,         REPS,EXIT
        byte    $20,"STREND",       hd,         STREND,EXIT

        byte    $20,"0",            hd,         _0,EXIT
        byte    $20,"1",            hd,         _1,EXIT
        byte    $20,"2",            hd,         _2,EXIT
        byte    $20,"3",            hd,         _3,EXIT
        byte    $20,"4",            hd,         _4,EXIT
'       byte    $20,"5",            hd,         _5,EXIT
'       byte    $20,"6",            hd,         _6,EXIT
'       byte    $20,"7",            hd,         _7,EXIT
        byte    $20,"8",            hd,         _8,EXIT
        byte    $20,"ON",           hd,         MINUS1,EXIT
        byte    $20,"TRUE",         hd,         MINUS1,EXIT
        byte    $20,"-1",           hd,         MINUS1,EXIT
        byte    $20,"BL",           hd,         BL,EXIT
        byte    $20,"16",           hd,         _16,EXIT
        byte    $20,"FALSE",        hd,         _0,EXIT
        byte    $20,"OFF",          hd,         _0,EXIT

        byte    $20,"1+",           hd,         INC,EXIT
        byte    $20,"1-",           hd,         DEC,EXIT
        byte    $20,"+",            hd,         PLUS,EXIT
        byte    $20,"-",            hd,         MINUS,EXIT

        byte    $20,"ADO",          hd,         ADO,EXIT
        byte    $20,"DO",           hd,         DO,EXIT
```

```
        byte    $20,"LOOP",             hd,         LOOP,EXIT
        byte    $20,"+LOOP",            hd,         PLOOP,EXIT
        byte    $20,"FOR",              hd,         FOR,EXIT
        byte    $20,"NEXT",             hd,         forNEXT,EXIT


        byte    $20,"INVERT",           hd,         INVERT,EXIT
        byte    $20,"AND",              hd,         _AND,EXIT
        byte    $20,"ANDN",             hd,         _ANDN,EXIT
        byte    $20,"OR",               hd,         _OR,EXIT
        byte    $20,"XOR",              hd,         _XOR,EXIT

        byte    $20,"ROL",              hd,         _ROL,EXIT
        byte    $20,"ROR",              hd,         _ROR,EXIT
        byte    $20,"SHR",              hd,         _SHR,EXIT
        byte    $20,"SHL",              hd,         _SHL,EXIT
'       byte    $20,"SHL16",            hd,         _SHL16,EXIT
        byte    $20,"2/",               hd,         _SHR1,EXIT
        byte    $20,"2*",               hd,         _SHL1,EXIT
        byte    $20,"REV",              hd,         _REV,EXIT
        byte    $20,"MASK",             hd,         MASK,EXIT
        byte    $20,">N",               hd,         toNIB,EXIT
        byte    $20,">B",               hd,         toBYTE,EXIT
'       byte    $20,">W",               hd,         toWORD,EXIT

        byte    $20,"0=",               hd,         ZEQ,EXIT
        byte    $20,"NOT",              hd,         ZEQ,EXIT
        byte    $20,"=",                hd,         EQ,EXIT
        byte    $20,">",                hd,         GT,EXIT

        byte    $20,"C@",               hd,         CFETCH,EXIT
        byte    $20,"W@",               hd,         WFETCH,EXIT
        byte    $20,"@",                hd,         FETCH,EXIT
        byte    $20,"C+!",              hd,         CPLUSST,EXIT
        byte    $20,"C!",               hd,         CSTORE,EXIT
        byte    $20,"C@++",             hd,         CFETCHINC,EXIT
        byte    $20,"W+!",              hd,         WPLUSST,EXIT
        byte    $20,"W!",               hd,         WSTORE,EXIT
        byte    $20,"+!",               hd,         PLUSST,EXIT
        byte    $20,"!",                hd,         STORE,EXIT
'       byte    $20,"BIT!",             hd,         BIT,EXIT
        byte    $20,"SET",              hd,         SET,EXIT
        byte    $20,"CLR",              hd,         CLR,EXIT


        byte    $20,"U/",               hd,         UDIVIDE,EXIT
        byte    $20,"U/MOD",            hd,         UDIVMOD,EXIT
        byte    $20,"UM*",              hd,         UMMUL,EXIT
        byte    $20,"ABS",              hd,         _ABS,EXIT
        byte    $20,"-NEGATE",          hd,         MNEGATE,EXIT
        byte    $20,"?NEGATE",          hd,         QNEGATE,EXIT
        byte    $20,"NEGATE",           hd,         NEGATE,EXIT

{
        byte    $20,"IN@",              hd,         PFETCH,EXIT
        byte    $20,"P@",               hd,         PFETCH,EXIT
        byte    $20,"OUT!",             hd,         PSTORE,EXIT
        byte    $20,"P!",               hd,         PSTORE,EXIT
        byte    $20,"STROBE",           hd,         STROBE,EXIT
}
        byte    $20,"CLOCK",            hd,         CLOCK,EXIT
        byte    $20,"OUTSET",           hd,         OUTSET,EXIT
        byte    $20,"OUTCLR",           hd,         OUTCLR,EXIT
        byte    $20,"OUTPUTS",          hd,         OUTPUTS,EXIT
        byte    $20,"INPUTS",           hd,         INPUTS,EXIT
'       byte    $20,"WAITLOW",          hd,         WAITLOW,EXIT
        byte    $20,"SHROUT",           hd,         SHROUT,EXIT
        byte    $20,"SHRINP",           hd,         SHRINP,EXIT


        byte    $20,"RESET",            hd,         RESET,EXIT
        byte    $20,"0EXIT",            hd,         ZEXIT,EXIT
        byte    $20,"EXIT",             hd,         EXIT,EXIT
        byte    $20,"NOP",              hd,         _NOP,EXIT
        byte    $20,"3DROP",            hd,         DROP3,EXIT
        byte    $20,"?DUP",             hd,         QDUP,EXIT
        byte    $20,"3RD",              hd,         THIRD,EXIT
        byte    $20,"4TH",              hd,         FOURTH,EXIT


        byte    $20,"CALL",             hd,         ACALL,EXIT
        byte    $20,"JUMP",             hd,         AJMP,EXIT
```

```
                byte    $20,"BRANCH>",          hd,             POPBRANCH,EXIT

                byte    $20,">R",               hd,             PUSHR,EXIT
                byte    $20,"R>",               hd,             RPOP,EXIT
                byte    $20,">L",               hd,             PUSHL,EXIT
                byte    $20,"L>",               hd,             LPOP,EXIT

                byte    $20,"REG",              hd,             ATREG,EXIT
                byte    $20,"LAP",              hd,             LAP,EXIT
'               byte    $20,"NEWCNT",           hd,             LAP,EXIT        ' alias for backward compatibilty

                byte    $20,"(WAITPEQ)",        hd,             _WAITPEQ,EXIT
                byte    $20,"(WAITPNE)",        hd,             _WAITPNE,EXIT
'               byte    $20,"(WAITHILO)",       hd,             WAITHILO,EXIT

{ we don't really need to have the names of these codes in the dictionary - useful for a disassembler though
                byte    $20,"(XCALL)",          hd+pr,          XCALL,EXIT
                byte    $20,"(YCALL)",          hd+pr,          YCALL,EXIT
                byte    $20,"(ELSE)",           hd+pr,          _ELSE,EXIT
                byte    $20,"(IF)",             hd+pr,          _IF,EXIT
                byte    $20,"(UNTIL)",          hd+pr,          _UNTIL,EXIT
                byte    $20,"(AGAIN)",          hd+pr,          _AGAIN,EXIT
                byte    $20,"(REG)",            hd+pr,          REG,EXIT
                byte    $20,"(PUSH4)",          hd+pr,          PUSH4,EXIT
                byte    $20,"(PUSH3)",          hd+pr,          PUSH3,EXIT
                byte    $20,"(PUSH2)",          hd+pr,          PUSH2,EXIT
                byte    $20,"(PUSH1)",          hd+pr,          PUSH1,EXIT
                byte    $20,"(VAR)",            hd+pr,          VARB,EXIT
}
                byte    $20,"I",                hd,             I,EXIT
                byte    $20,"SPIWRB",           hd,             SPIWRB,EXIT
                byte    $20,"SPIWR16",          hd,             SPIWR16,EXIT
                byte    $20,"SPIWR",            hd,             SPIWR,EXIT
                byte    $20,"SPIWRX",           hd,             SPIWRX,EXIT
                byte    $20,"SPIRD",            hd,             SPIRD,EXIT
                byte    $20,"SPIRDX",           hd,             SPIRDX,EXIT

                byte    $20,"(OPCODE)",         hd,             OPCODE,EXIT



' Extended operation - accesses high 256 longs of VM cog

                byte    $20,"CMOVE",            hd+xc+sq,       XOP,pCMOVE
                byte    $20,"(EMIT)",           hd+xc+sq,       XOP,pEMIT
                byte    $20,"CMPSTR",           hd+xc+sq,       XOP,pCMPSTR
                byte    $20,"LOADMOD",          hd+xc+sq,       XOP,pLOADMOD
                byte    $20,"RUNMOD",           hd+xc+sq,       XOP,pRUNMOD
                byte    $20,"COGID",            hd+xc+sq,       XOP,_COGID
                byte    $20,"!RP",              hd+xc+sq,       XOP,pInitRP
'               byte    $20,"SPR@",             hd+xc+sq,       XOP,pSPRFETCH
                byte    $20,"COG@",             hd+xc+sq,       XOP,pCOGFETCH
                byte    $20,"COGREG",           hd+xc+sq,       XOP,pCOGREG
                byte    $20,"COG!",             hd+xc+sq,       XOP,pCOGSTORE
                byte    $20,"PASM",             hd+xc+sq,       XOP,pPASM
                byte    $20,"MYOP",             hd+xc+sq,       XOP,pMYOP
                byte    $20,"LSTACK",           hd+xc+sq,       XOP,pLSTACK
                byte    $20,"DELTA",            hd+xc+sq,       XOP,pDELTA
                byte    $20,"WAITCNT",          hd+xc+sq,       XOP,pWAITCNTS

' Two instruction sequences
'

                byte    $20,"2+",               hd+xc+sq,       _2,PLUS
                byte    $20,"2-",               hd+xc+sq,       _2,MINUS
                byte    $20,"2DUP",             hd+xc+sq,       OVER,OVER
                byte    $20,"*",                hd+xc+sq,       UMMUL,DROP
                byte    $20,"0<>",              hd+xc+sq,       ZEQ,ZEQ
                byte    $20,"<>",               hd+xc+sq,       EQ,ZEQ
```

{ INTERPRETED BYTECODE HEADERS }

```
                byte    $20,"!SP",              hd+xc,          XCALL,xInitStack
                byte    $20,"SP!",              hd+xc,          XCALL,xSPSTORE
                byte    $20,"DEPTH",            hd+xc,          XCALL,xDEPTH

                byte    $20,"(:)",              hd+xc,          XCALL,x_COLON
                byte    $20,":",                hd+xc+im,       XCALL,xCOLON
                byte    $20,"pub",              hd+xc+im,       XCALL,xPUBCOLON
                byte    $20,"pri",              hd+xc+im,       XCALL,xPRIVATE
```

```
byte    $20,"UNSMUDGE",         hd+xc+im,       XCALL,xUNSMUDGE

byte    $20,"IF",               hd+xc+im,       XCALL,x_IF_
byte    $20,"ELSE",             hd+xc+im,       XCALL,x_ELSE_
byte    $20,"THEN",             hd+xc+im,       XCALL,x_THEN_
byte    $20,"ENDIF",            hd+xc+im,       XCALL,x_THEN_
byte    $20,"BEGIN",            hd+xc+im,       XCALL,x_BEGIN_
byte    $20,"UNTIL",            hd+xc+im,       XCALL,x_UNTIL_
byte    $20,"AGAIN",            hd+xc+im,       XCALL,x_AGAIN_
byte    $20,"WHILE",            hd+xc+im,       XCALL,x_IF_
byte    $20,"REPEAT",           hd+xc+im,       XCALL,x_REPEAT_
byte    $20,";",                hd+xc+im,       XCALL,xENDCOLON

byte    $20,"\",                hd+xc+im,       XCALL,xCOMMENT
byte    $20,"""",               hd+xc+im,       XCALL,xCOMMENT
byte    $20,"(",                hd+xc+im,       XCALL,xBRACE
byte    $20,"{",                hd+xc+im,       XCALL,xCURLY
byte    $20,"}",                hd+xc+im,       _NOP,EXIT
byte    $20,"IFNDEF",           hd+xc+im,       XCALL,xIFNDEF
byte    $20,"IFDEF",            hd+xc+im,       XCALL,xIFDEF
byte    $20,$22,                hd+xc+im,       XCALL,x_STR_
byte    $20,$2E,$22,            hd+xc+im,       XCALL,x_PSTR_
byte    $20,"(",$2E,$22,")",    hd+xc+im,       XCALL,xPRTSTR

byte    $20,"TO",               hd+xc+im,       XCALL,xATO

byte    $20,"COMPILE",          hd+xc+im,       XCALL,xCOMPILE
byte    $20,"BCOMP",            hd+xc+im,       XCALL,xBCOMP
byte    $20,"C,",               hd+xc+im,       XCALL,xCCOMP
byte    $20,"|",                hd+xc+im,       XCALL,xCCOMP
byte    $20,"||",               hd+xc+im,       XCALL,xWCOMP
byte    $20,",",                hd+xc+im,       XCALL,xLCOMP


byte    $20,"BREAK",            hd+xc+im,       XCALL,xISEND
byte    $20,"CASE",             hd+xc+im,       XCALL,xIS
byte    $20,"SWITCH",           hd+xc,          XCALL,xSWITCH
byte    $20,"SWITCH@",          hd+xc,          XCALL,xSWITCHFETCH
byte    $20,"SWITCH=",          hd+xc,          XCALL,xISEQ
byte    $20,"SWITCH><",         hd+xc,          XCALL,xISWITHIN

byte    $20,"STACKS",           hd+xc,          XCALL,xSTACKS
byte    $20,"XCALLS",           hd+xc,          XCALL,xXCALLS
byte    $20,"REBOOT",           hd+xc,          XCALL,xREBOOT
byte    $20,"STOP",             hd+xc,          XCALL,xSTOP

byte    $20,"[SSD]",            hd+xc,          XCALL,xSSD
byte    $20,"[ESPIO]",          hd+xc,          XCALL,xESPIO
byte    $20,"[SPIO]",           hd+xc,          XCALL,xSPIO
byte    $20,"[SPIOD]",          hd+xc,          XCALL,xSPIOD
byte    $20,"[SDRD]",           hd+xc,          XCALL,xSDRD
byte    $20,"[SDWR]",           hd+xc,          XCALL,xSDWR
byte    $20,"[PWM32]",          hd+xc,          XCALL,xPWM32
byte    $20,"[PWM32!]",         hd+xc,          XCALL,xPWMST32
byte    $20,"[PLOT]",           hd+xc,          XCALL,xPLOT
byte    $20,"BCA",              hd+xc,          XCALL,xBCA
'       byte    $20,"[CMPSTR]",         hd+xc,          XCALL,xCMPSTRMOD

        byte    $20,"SET?",             hd+xc,          XCALL,xSETQ

        byte    $20,"0<",               hd+xc,          XCALL,xZLT
        byte    $20,"<",                hd+xc,          XCALL,xLT
        byte    $20,"U<",               hd+xc,          XCALL,xULT
        byte    $20,"WITHIN",           hd+xc,          XCALL,xWITHIN
        byte    $20,"?EXIT",            hd+xc,          XCALL,xIFEXIT

        byte    $20,"ERASE",            hd+xc,          XCALL,xERASE
        byte    $20,"FILL",             hd+xc,          XCALL,xFILL
        byte    $20,"ms",               hd+xc,          XCALL,xms
        byte    $20,"us",               hd+xc,          XCALL,xus

        byte    $20,"READBUF",          hd+xc,          XCALL,xREADBUF
        byte    $20,"KEY",              hd+xc,          XCALL,xKEY
        byte    $20,"WKEY",             hd+xc,          XCALL,xWKEY
        byte    $20,"(KEY)",            hd+xc,          XCALL,xCONKEY

        byte    $20,"HEX",              hd+xc,          XCALL,xHEX
        byte    $20,"DECIMAL",          hd+xc,          XCALL,xDECIMAL
        byte    $20,"BINARY",           hd+xc,          XCALL,xBIN


        byte    $20,".S",               hd+xc,          XCALL,xPRTSTK
```

```
        byte    $20,"HDUMP",            hd+xc,          XCALL,xDUMP
        byte    $20,"COGDUMP",          hd+xc,          XCALL,xCOGDUMP

        byte    $20,".STACKS",          hd+xc,          XCALL,xPRTSTKS
        byte    $20,"DEBUG",            hd+xc,          XCALL,xDEBUG
        byte    $20,"EMIT",             hd+xc,          XCALL,xEMIT
'       byte    $20,"?EMIT",            hd+xc,          XCALL,xQEMIT
        byte    $20,"CLS",              hd+xc,          XCALL,xCLS
        byte    $20,"SPACE",            hd+xc,          XCALL,xSPACE
        byte    $20,"BELL",             hd+xc,          XCALL,xBELL
        byte    $20,"CR",               hd+xc,          XCALL,xCR
        byte    $20,"SPINNER",          hd+xc,          XCALL,xSPINNER
        byte    $20,".HEX",             hd+xc,          XCALL,xPRTHEX
        byte    $20,".BYTE",            hd+xc,          XCALL,xPRTBYTE
        byte    $20,".WORD",            hd+xc,          XCALL,xPRTWORD
        byte    $20,".LONG",            hd+xc,          XCALL,xPRTLONG
        byte    $20,".",                hd+xc,          XCALL,xPRT
        byte    $20,">DIGIT",           hd+xc,          XCALL,xTODIGIT
        byte    $20,"NUMBER",           hd+xc,          XCALL,xNUMBER
'       byte    $20,">UPPER",           hd+xc,          XCALL,xTOUPPER
        byte    $20,"SCRUB",            hd+xc,          XCALL,xSCRUB
        byte    $20,"GETWORD",          hd+xc,          XCALL,xGETWORD
        byte    $20,"SEARCH",           hd+xc,          XCALL,xSEARCH
        byte    $20,"FINDSTR",          hd+xc,          XCALL,xFINDSTR
'       byte    $20,"CMPSTR",           hd+xc,          XCALL,xCMPSTR
        byte    $20,"NFA>CFA",          hd+xc,          XCALL,xNFACFA
        byte    $20,"EXECUTE",          hd+xc,          XCALL,xEXECUTE

        byte    $20,"VER",              hd+xc,          XCALL,xGETVER
        byte    $20,".VER",             hd+xc,          XCALL,xPRTVER
        byte    $20,"TACHYON",          hd+xc,          XCALL,x_TACHYON

        byte    $20,"@PAD",             hd+xc,          XCALL,xATPAD
        byte    $20,"HOLD",             hd+xc,          XCALL,xHOLD
        byte    $20,">CHAR",            hd+xc,          XCALL,xTOCHAR
        byte    $20,"#>",               hd+xc,          XCALL,xRHASH
        byte    $20,"<#",               hd+xc,          XCALL,xLHASH
        byte    $20,"#",                hd+xc,          XCALL,xHASH
        byte    $20,"#S",               hd+xc,          XCALL,xHASHS
        byte    $20,"PRINT$",           hd+xc,          XCALL,xPSTR
        byte    $20,"LEN$",             hd+xc,          XCALL,xSTRLEN
        byte    $20,"U.",               hd+xc,          XCALL,xUPRT
        byte    $20,".DEC",             hd+xc,          XCALL,xPRTDEC
        byte    $20,"DISCARD",          hd+xc,          XCALL,xDISCARD
        byte    $20,"COGINIT",          hd+xc,          XCALL,xCOGINIT

        byte    $20,"<CMOVE",           hd+xc,          XCALL,xRCMOVE



' TASK REGISTERS

'       byte    $20,"REG",              hd+xc,          REG,0           '
        byte    $20,"flags",            hd+xc,          REG,flags
        byte    $20,"base",             hd+xc,          REG,base
        byte    $20,"digits",           hd+xc,          REG,digits
        byte    $20,"delim",            hd+xc,          REG,delim
        byte    $20,"word",             hd+xc,          REG,wordbuf
        byte    $20,"switch",           hd+xc,          REG,uswitch
        byte    $20,"autorun",          hd+xc,          REG,autovec
        byte    $20,"keypoll",          hd+xc,          REG,keypoll
        byte    $20,"tasks",            hd+xc,          REG,tasks
        byte    $20,"unum",             hd+xc,          REG,unum                ' user number processing
        byte    $20,"uemit",            hd+xc,          REG,uemit
        byte    $20,"ukey",             hd+xc,          REG,ukey
        byte    $20,"names",            hd+xc,          REG,names
        byte    $20,"here",             hd+xc,          REG,here
        byte    $20,"codes",            hd+xc,          REG,codes
        byte    $20,"errors",           hd+xc,          REG,errors
        byte    $20,"baudcnt",          hd+xc,          REG,baudcnt
        byte    $20,"prompt",           hd+xc,          REG,prompt
        byte    $20,"ufind",            hd+xc,          REG,findvec             ' user find method
        byte    $20,"create",           hd+xc,          REG,createvec           ' user CREATE
        byte    $20,"lines",            hd+xc,          REG,linenum
        byte    $20,"lastkey",          hd+xc,          REG,lastkey


        byte    $20,"ALLOT",            hd+xc,          XCALL,xALLOT
        byte    $20,"ALLOCATED",        hd+xc,          XCALL,xALLOCATED
        byte    $20,"HERE",             hd+xc,          XCALL,xHERE
        byte    $20,"AUTO!",            hd+xc,          XCALL,xAUTOST
        byte    $20,">VEC",             hd+xc,          XCALL,xTOVEC
```

```
            byte    $20,">PFA",             hd+xc,          XCALL,xPFA
            byte    $20,"[NFA']",           hd+xc,          XCALL,xNFATICK
            byte    $20,"[']",              hd+xc,          XCALL,xTICK
            byte    $20,"ERROR",            hd+xc,          XCALL,xERROR
            byte    $20,"NOTFOUND",         hd+xc,          XCALL,xNOTFOUND

' IMMEDIATE
            byte    $20,"NFA'",             hd+xc+im,       XCALL,x_NFATICK
            byte    $20,"'",                hd+xc+im,       XCALL,xATICK

            byte    $20,"AUTORUN",          hd+xc+im,       XCALL,xAUTORUN

' Building words
            byte    $20,"[COMPILE]",        hd+xc+im,       XCALL,xCOMPILES
            byte    $20,"GRAB",             hd+xc+im,       XCALL,xGRAB
            byte    $20,"LITERAL",          hd+xc+im,       XCALL,xLITCOMP

            byte    $20,"(CREATE)",         hd+xc,          XCALL,xCREATESTR
            byte    $20,"CREATEWORD",       hd+xc+im,       XCALL,xCREATEWORD
            byte    $20,"CREATE",           hd+xc+im,       XCALL,xCREATE

            byte    $20,"TASK",             hd+xc,          XCALL,xTASK
            byte    $20,"IDLE",             hd+xc,          XCALL,xIDLE
            byte    $20,"LOOKUP",           hd+xc,          XCALL,xVECTORS
'           byte    $20,"VECTORS",          hd+xc,          XCALL,xVECTORS
            byte    $20,"+CALL",            hd+xc,          XCALL,xAddACALL
            byte    $20,"BUFFERS",          hd+xc,          XCALL,xBUFFERS
            byte    $20,"FREE",             hd+xc,          XCALL,xFREE
            byte    $20,"TERMINAL",         hd+xc,          XCALL,xTERMINAL
            byte    $20,"CONSOLE",          hd+xc,          XCALL,xCONSOLE
            byte    $20,"KOLD",             hd+xc,          XCALL,xCOLDST           'renamed to avoid conflict

            byte    $20,"*end*",            hd+xc+sq,       _NOP,_NOP           ' dummy used to find the end


enddict     byte    0,0,0       ' Mark the end of the kernel dictionary (2nd and 3rd byte is a pointer or null)

{
                                ****************************************
                                ************* COG IMAGE & BUFFERS *************
                                ****************************************
}



                        org         $10
s                       ' just an offset to be used in DAT sections rather than the distracting +$10


' Roundabout way in Spin compiler to align this area to the next 256 byte boundary
aln1        byte    0[$100-(@aln1+s-((@aln1+s)&$FF00))]
```

{ TACHYON VM - COG KERNEL (PASM) }

```
DAT

{{
Byte tokens directly address code in the first 256 longs of the cog.
A two byte-code instruction XOP allows access to the second 256 longs
Rather than a jump table most functions are short or cascaded to optimize COG memory
Larger fragments of code jump to the second half of the cog's memory.
As a result of not using a jump table (there's not enough memory) there are gaps
in the bytecode values and not all values are usable.

~~The formatted source has bytecode instruction labels as bold white on red background.~~
}}

                        org         0

RESET                   mov         IP,PAR                  ' Load the IP with the address of the first instruction as if it were an XOP

' position XOP here so that any search for an address of an XOP word returns with the correct cog address of $01xx
' Use next byte as an opcode that directly addresses top 256 words of cog
XOP                     rdbyte      instr,IP                ' get next bytecode
                        or          instr,#$100             ' shift range
                        jmp         #doNext+1               ' IP++, execute
```

{ * RUNTIME BYTECODE INTERPRETER *** }**

```
'                   *               *           *                                   *
' Fetch the next byte code instruction in hub RAM pointed to by the instruction pointer IP
```

```
' This is the very heart of the runtime interpreter
'
doNEXT                  rdbyte      instr,IP                'read byte code instruction
                        add         IP,#1 wc                'advance IP to next byte token (clears the carry too!)
                        jmp         instr                   'execute the code by directly indexing the first 256 long in cog



' Find the end of the string which could end in a null or any characeter >$7F
' this is also used to find the end of a larger text buffer
' STREND ( ptr -- ptr2 )
STREND
fchlp                   rdbyte      R0,tos                  ' read a byte
                        sub         R0,#1                   ' end is either a null or anything >$7F
                        cmp         R0,#$7E wc
                if_c    add         tos,#1
                if_c    jmp         #fchlp
                        jmp         unext


' 0EXIT ( flg -- ) Exit if flg is false (or zero)  Used in place of IF.....THEN EXIT as false would just end up exiting
ZEXIT                   call        #POPX
                        tjnz        X,unext
'
' EXIT a bytecode definition by popping the top of the return stack into the IP
EXIT                    call        #RPOPX                  ' Pop from return stack into X
JUMPX                   mov         IP,X                    ' update IP
_NOP                    jmp         unext                   ' continue
```

```
                                                { *** STACK OPERATORS *** }

' DROP3 ( n1 n2 n3 -- )  Pop the top 3 items off the datastack and discard them (used mostly by cog kernel)
DROP3                   call        #POPX
' DROP2 ( n1 n2 -- )  Pop the top 2 items off the datastack and discard them
DROP2                   call        #POPX
' 1us execution time including bytecode read and execute
' DROP ( n1 -- )  Pop the top item off the datastack and discard it
DROP                    call        #POPX
                        jmp         unext

' ?DUP ( n1 -- n1 n1 | 0 ) DUP n1 if non-zero
QDUP                    tjz         tos,unext
' DUP ( n1 - n1 n1 ) Duplicate the top item on the stack
DUP                     mov         X,tos                   ' Read directly from the top of the data stack
PUSHX                   call        #_PUSHX                 ' Push the internal X register onto the datastack
                        jmp         unext

' OVER ( n1 n2 -- n1 n2 n1 )
OVER                    mov         X,tos+1             'read second data item and push
                        jmp         #PUSHX
' 3RD ( n1 n2 n3 -- n1 n2 n3 n1 ) Copy the 3rd item onto the stack
THIRD                   mov         X,tos+2             ' read third data item
                        jmp         #PUSHX
' 4TH ( n1 n2 n3 n4 -- n1 n2 n3 n4 n1 ) Copy the 4th item onto the stack
FOURTH                  mov         X,tos+3
                        jmp         #PUSHX

' BOUNDS ( n1 n2 -- n2+n1 n1 ) == OVER + SWAP
BOUNDS                  add         tos,tos+1
' SWAP ( n1 n2 -- n2 n1 ) Swap the top two items
SWAP                    mov         X,tos+1
SWAPX                   mov         tos+1,tos
PUTX                    mov         tos,X
                        jmp         unext
' ROT ( a b c -- b c a )
ROT                     mov         X,tos+2
                        mov         tos+2,tos+1
                        jmp         #SWAPX



                                                { *** ARITHMETIC *** }

' - ( n1 n2 -- n3 ) Subtract n2 from n1
MINUS                   neg         tos,tos                 ' (note: save one long by negating and adding)
' + ( n1 n2 -- n3 ) Add top two stack items together and replace with result
PLUS                    add         tos+1,tos
                        jmp         #DROP

' 1- ( n1 -- n1-1 )
DEC                     test        $,#1 wc
' 1+ ( n1 -- n1+1 )
INC                     sumc        tos,#1                  ' inc or dec depending upon carry (default cleared by doNEXT)
                        jmp         unext


' -NEGATE ( n1 sn -- n1 | -n1 ) negate n1 if the sign of sn is negative (used in signed divide op)
MNEGATE                 shr         tos,#31
 ' ?NEGATE ( n1 flg -- n2 ) negate n1 if flg is true
QNEGATE                 tjz         tos,#DROP
```

```
                                call    #POPX
' NEGATE ( n1 -- n2 )  equivalent to  n2 = 0-n1
NEGATE                          neg     tos,tos
                                jmp     unext

' u/mod ( u1 u2 -- remainder quotient) both remainder and quotient are 32 bit unsigned numbers
UDIVMOD                         call    #_UDIVMOD
                                jmp     unext

' U/ ( n1 n2 -- n3 ) unsigned divide
UDIVIDE                         call    #_UDIVMOD
NIP                             mov     tos+1,tos
                                jmp     #DROP



                                                                { *** BOOLEAN *** }

' 400ns execution time including bytecode read and execute
' INVERT ( n1 -- n2 ) bitwise invert n1 and replace with result n2
INVERT                          add     tos,#1
                                jmp     #NEGATE
{
_BITS                           test    $,#1 wc                 ' set carry
                                rcl     ACC,tos
                                and     tos+1,ACC
                                jmp     #DROP
}
_AND                            and     tos+1,tos
                                jmp     #DROP
_ANDN                           andn    tos+1,tos
                                jmp     #DROP
_OR                             or      tos+1,tos
                                jmp     #DROP
_XOR                            xor     tos+1,tos
                                jmp     #DROP
' 1.2us execution time including bytecode read and execute
' SHR ( n1 cnt -- n2 ) Shift n1 right by count (5 lsbs )
_SHR                            shr     tos+1,tos
                                jmp     #DROP
_SHL                            shl     tos+1,tos
                                jmp     #DROP
_ROL                            rol     tos+1,tos
                                jmp     #DROP
_ROR                            ror     tos+1,tos
                                jmp     #DROP

' 400ns execution time including bytecode read and execute
' 2/ ( n1 -- n1 ) shift n1 right one bit (equiv to divide by 2)
_SHR1                           shr     tos,#1
                                jmp     unext
'_SHL16                         shl     tos,#15
' 2* ( n1 -- n2 ) shift n1 left one bit (equiv to multiply by 2)
_SHL1                           shl     tos,#1
                                jmp     unext

' REV ( n1 bits -- n2 ) Reverse LSBs of n1 and zero-extend
_REV                            rev     tos+1,tos
                                jmp     #DROP

' 400ns execution time including bytecode read and execute
' MASK ( bitpos -- bitmask  \ only the lower 5 bits of bitpos are taken, regardless of the higher bits )
MASK                            mov     X,tos
                                mov     tos,#1
                                shl     tos,X
                                jmp     unext

' >N ( n -- nibble ) mask n to a nibble
toNIB                           and     tos,#$0F
' >B ( n -- nibble ) mask n to a byte
toBYTE                          and     tos,#$FF
                                jmp     unext



                                                                { *** COMPARISON *** }

' Basic instructions from which other comparison instructions are built from

' = ( n1 n2 -- flg ) true if n1 is equal to n2
EQ                              sub     tos+1,tos               ' n1 == 0 if equal
                                call    #POPX                   ' drop n2
'                               ------------
' 0= ( n1 -- flg ) true if n1 equals 0 - same as a boolean NOT where TRUE becomes FALSE
_NOT
ZEQ                             cmp     tos,#1 wc               ' kuroneko method, nice and neat
SETZ                            subx    tos, tos                ' a carry becomes -1, else 0
                                jmp     unext

' > ( n1 n2 -- flg ) true if n1 > n2
GT                              cmps    tos,tos+1  wc           ' n1 > n2: carry set
```

```
                        subx    tos+1,tos+1
                        jmp     #DROP



                                                        { *** MEMORY *** }

' C@++  ( caddr -- caddr+1 byte ) fetch byte character and increment address
CFETCHINC               mov     X,tos                   ' dup the address
                        call    #_PUSHX
                        add     tos+1,#1                ' inc the backup address
' C@  ( caddr -- byte ) Fetch a byte from hub memory
CFETCH                  rdbyte  tos,tos
                        jmp     unext

' W@  ( waddr -- word ) Fetch a word from hub memory
WFETCH                  rdword  tos,tos
                        jmp     unext

' @  ( addr -- long ) Fetch a long from hub memory
FETCH                   rdlong  tos,tos
                        jmp     unext

' C+!  ( n caddr -- ) add n to byte at hub addr
CPLUSST                 rdbyte  X,tos                   ' read in word from adress
                        add     tos+1,X                 ' add to contents of address - cascade
' C!  ( n caddr -- ) store n to byte at addr
CSTORE                  wrbyte  tos+1,tos               ' write the byte using address on the tos
                        jmp     #DROP2

' W+!  ( n waddr -- ) add n to word at hub addr
WPLUSST                 rdword  X,tos                   ' read in word from address
                        add     tos+1,X
' W!  ( n waddr -- ) store n to word at addr
WSTORE                  wrword  tos+1,tos
                        jmp     #DROP2

' +!  ( n addr -- ) add n to long at hub addr
PLUSST                  rdlong  X,tos                   ' read in long from address
                        add     tos+1,X
' !  ( n addr -- ) store n to long at addr
STORE                   wrlong  tos+1,tos
                        jmp     #DROP2

' BIT! ( mask caddr state -- ) Set or clear bit(s) in hub byte
'BIT                    call    #POPX
'                       tjz     X,#CLR                  ' carry clear, finalize
' SET ( mask caddr -- ) Set bit(s) in hub byte
SET                     test    $,#1  wc                ' set the carry flag
' Finalize the bit operation by read/writing the result
' ( mask caddr -- )
CLR                     rdbyte  X,tos                   ' Read the contents of the memory location
                        muxc    X,tos+1                 ' set or clear the bit(s) specd by mask
                        wrbyte  X,tos                   ' update
                        jmp     #DROP2



                                                        { *** LITERALS *** }

' LITERALS are stored unaligned in big endian format which faciliates cascading byte reads to accumulate the full number

' 3.6us execution time including bytecode read and execute
' ( -- 32bits ) Push a 32-bit literal onto the datastack by reading in the next 4 bytes (non-aligned)
_LONG
PUSH4                   call    #ACCBYTE                ' read the next byte @IP++ and shift accumulate
' 3us execution time including bytecode read and execute
' ( -- 24bits ) Push a 24-bit literal onto the datastack by reading in the next 3 bytes (non-aligned)
PUSH3                   call    #ACCBYTE
_WORD
' 2.4us execution time including bytecode read and execute
' ( -- 16bits) Push a 16-bit literal onto the datastack by reading in the next 2 bytes (non-aligned)
PUSH2                   call    #ACCBYTE
' 1.8us execution time including bytecode read and execute
' ( -- 8bits ) Push an 8-bit literal onto the datastack by reading in the next byte
_BYTE
PUSH1                   call    #ACCBYTE
PUSHACC                 call    #_PUSHACC               ' Push the accumulator onto the stack then zero it
                        jmp     unext



                                                        { *** FAST CONSTANTS *** }


' Push a preset literal onto the stack using just one bytecode
' Use the "accumulator" to push the value which is built up by incrementing and/or decrementing
' There is a minor penalty for the larger constants but it's still faster and more compact
' overall than using the PUSH1 method or the mov X,# method

' 140606 just reordered to 1 4 2 3 according to BCA results
```

```
' 140603 new method to allow any value in any order, relies on carry being cleared in doNEXT and min will always set carry here
BL          if_nc           min         ACC,#32+1 wc                ' 1.52us
_16         if_nc           min         ACC,#16+1 wc
_8          if_nc           min         ACC,#8+1 wc
_4          if_nc           min         ACC,#4+1 wc
_2          if_nc           min         ACC,#2+1 wc
_1          if_nc           min         ACC,#1+1 wc
_3          if_nc           min         ACC,#3+1 wc                 ' bytecode analysis reveals 3 is used quite heavily
_TRUE
MINUS1                      sub         ACC,#1
_FALSE
_0                          jmp         #PUSHACC                    ' 1.12us



                                                    { *** CONSTANTS & VARIABLES *** }
' Constants and variables etc are standalone fragments preceded by an opcode then the parameters, either a long or the addess of the parameter field

' Long aligned constant - created with CONSTANT and already aligned
CONL
                            rdlong      X,IP                        ' get constant
                            jmp         #PUSHX_EXIT


' Byte aligned variables start with this single byte code which returns with the address of the byte variable following
' long variables just externally align this opcode a byte before the boundary
' INLINE:
VARB                        mov         X,IP
PUSHX_EXIT                  call        #_PUSHX                     ' push address of variable
                            jmp         #EXIT

' OPCODE assumes that a long aligned long follows which contains the 32-bit opcode.
OPCODE                      rdlong      opc,IP                      ' read the long that follows (just like a constant)
                            nop
opc                         nop
                            jmp         #EXIT                       ' return back to caller



                                                    { *** I/O ACCESS *** }
{ not used - removed to extensions using COG@ COG!
' P@ ( -- n1 ) Read the input port A (assume it is always A for Prop 1)
PFETCH                      mov         X,INA
                            jmp         #PUSHX
' P! ( n1 -- ) Store n1 to the output port A
PSTORE                      mov         OUTA,tos
                            jmp         #DROP

' STROBE ( iomask -- ) Generate a 100ns low pulse - pins must be preset as outputs (first up anyway)
STROBE                      andn        OUTA,tos                    ' strobe low
                            jmp         #OUTSET                     ' release high (use jmp to add one extra cycle)

}

' CLOCK ( COGREG4=iomask ) Toggle multiple bits on the output)
CLOCK                       xor         OUTA,clockpins
                            jmp         unext

' OUTCLR ( iomask -- ) Clear multiple NUMBERbits on the output
OUTCLR                      andn        OUTA,tos
                            jmp         #OUTPUTS
' OUTMASK ( data iomask -- )
'                           call        #POPX
                            andn        OUTA,X                      ' clear all iomask outputs
' OUTSET ( iomask -- ) Set multiple bits on the output
OUTSET                      or          OUTA,tos
 ' OUTPUTS ( iomask -- ) Set selected port pins to outputs
OUTPUTS                     or          DIRA,tos
                            jmp         #DROP


' INPUTS ( iomask -- ) Set selected port pins to inputs
INPUTS                      andn        DIRA,tos
                            jmp         #DROP


WAITHILO                    waitpeq     reg3,reg3                   ' wait for a hi to lo - look for falling edge
' WAITPNE   Wait until input is low - REG3 = mask, REG0 = CNT
_WAITPNE                    waitpne     reg3,reg3                   ' use COGREG3 as the mask
                            mov         reg0,cnt                    ' capture count in COGREG0
                            jmp         unext

' WAITPEQ           Wait until input is high - REG3 = mask, REG1 = CNT
_WAITPEQ                    waitpeq     reg3,reg3
                            mov         reg1,cnt                    ' capture count in COGREG1
                            jmp         unext
```

```
                                                    { *** SERIAL I/O OPERATORS *** }
{
To maximize the speed of I/O operations especially serial I/O such as ASYNCH, I2C and SPI etc there are special
operators that avoid pushing and popping the stack and instead perform the I/O bit by bit and leave the
latest shifted version of the data on the stack.
}

' SHIFT from INPUT - Assembles with last bit received as msb - needs SHR to right justify if asynch data
' SHRINP ( iomask dat -- iomask dat/2 )
SHRINP                  test        tos+1,INA wc
                        rcr         tos,#1
                        jmp         unext


{ SHIFT to OUT -
This is optimized for when you are sending out multiple bits as in asynchronous serial data or I2C
Shift data one bit right into output via iomask - leave mask & shifted data on stack (looping)
400ns execution time including bytecode read and execute  or 200ns/bit with REPS }
' SHROUT ( iomask dat -- iomask dat/2 )
SHROUT                  shr         tos,#1 wc                   ' Shift right and get lsb
                        muxc        OUTA,tos+1                  ' reflect state to output
                        jmp         unext


                                                                { *** SPI *** }
' SPI INSTRUCTIONS
' Simple fast, bare-bones SPI - transmits 8 bits MSB first - data must be left justified - data is not discarded
' Usage: $1234 16 SHL SPIWR SPIWR 'transmit 16 bits ( 140928 just SPIWR16 )
' SPIWRX permits variable number of bits if spicnt is set directly with @SPICNT COGREG!
' SPIWR ( data -- data<<8 )
' SPIWRX could be an instruction if we manually set the spicnt beforehand
SPIWR16                 rol         tos,#24
                        mov         ACC,#8                      ' force a 16-bit transfer
SPIWRB                  rol         tos,#24                     ' left justify and write byte to SPI (2.8us)
SPIWR                   add         ACC,#8                      ' code execution time of 2.25us + overhead
SPIWRX                  andn        outa,spice                  ' always activate the chip enable (saves time in HL)
SPIwrlp                 rol         tos,#1 wc                   ' assume msb first (140208 update now rotates)
                        muxc        OUTA,spiout                 ' send next bit of data out
                        xor         OUTA,spisck                 ' toggle clock
                        xor         OUTA,spisck                 ' toggle clock
                        djnz        ACC,#SPIwrlp
                        jmp         unext


' Receive data and shift into existing stack parameter
' If MOSI needs to be in a certain state then this should be set beforehand
' Clock will simply pulse from it's starting state - normally low
' Useage:  0 SPIRD SPIRD SPIRD SPIRD     'read 32-bits as one long
' SPIRD ( data -- data<<8 )              ' 2.8us
SPIRD                   mov         spicnt,#8                   ' always read back 8-bits
SPIRDX                  andn        outa,spice                  ' always activate the chip enable (saves time in HL)
SPIrdlp                 xor         OUTA,spisck                 ' toggle clock
                        test        spiinp,INA wc               ' read data from device
                        rcl         tos,#1
                        xor         OUTA,spisck                 ' toggle clock
                        djnz        spicnt,#SPIrdlp
                        jmp         unext


' I ( -- index ) The current loop index is at a fixed address just under the loop limit at top of an ascending loop stack
I                       mov         X,loopstk+1
                        jmp         #PUSHX


                                                                { *** BRANCH & LOOP *** }

' ACALL ( adr -- ) Call arbitrary (from the stack) address - used to execute user vectors
ACALL                   call        #SAVEIP                     ' save current IP onto return stack
AJMP                    mov         IP,tos                      ' jump to address on top of the data stack
                        jmp         #DROP
{
 Since Tachyon uses bytecodes for instructions it also uses bytes to address code.
 Since bytes are limited to 256 values these are used instead to lookup the absolute address of the code
 from a table of vectors. To extend that beyond 256 values there are further opcodes which index the  table for a total of 1,024 vectors.

 Vectored call instructions form a 10-bit index with the upper 2 bits derived from the instruction so calls to other words only use 2 bytes total
Note: Originally there was only an XCALL but new opcodes were added to expand the vector table  however the compiled kernel only uses XCALL
}
VCALL                   mov         ACC,#2                      ' high word of upper page
ZCALL                   add         ACC,zcalls                  ' low word of upper page

' Perform a call to kernel bytecode via the XCALLS but reusing the high word of each vector
' The YCALLs are implemented by the runtime compiler to fully utilize the XCALLs table (high word of longs)
YCALL                   add         ACC,#2
' Inline call to kernel bytecode via the XCALLS vector table using the following inline byte as an index
```

```
XCALL                   add     ACC,Xptr                ' ACC = vector table offset ( lopage,loword -> hipage,hiword)
xycall                  call    #SETUPIP                ' Save IP and read next bytecode into X (offset in table)
                        shl     X,#2                    ' offset into longs in hub RAM
                        add     X,ACC                   ' Add to vector table base,now points to vector
                        rdword  IP,X                    ' Load IP with vector, now points to bytecode
ACC0                    mov     ACC,#0                  ' Always clear ACC to zero ready for reuse (repos for hub access)
                        jmp     unext

' Read the next byte as a negative displacement and jump back
_AGAIN
                        test    $,#1 wc                 ' setc for negative displacement
' Jump forward by reading the next byte inline and adding that to the IP (at that point)
JUMP
_ELSE           call    #GETBYTE                        ' read the forward displacement byte
                        sumc    IP,X                    ' jump to that forward location
                        jmp     unext

' If flg is zero than jump forward by reading the next byte inline and adding that to the IP (at that point)
' IF R( flg -- )
' Read the next byte as a positive displacement and branch forward
JZBACK
_UNTIL                  test    $,#1 wc                 'set carry for negative branch
JZ
_IF                     call    #GETBYTE                'read in next byte at IP and inc IP
JMPIF                   tjnz    tos,#DROP               'test flag on stack - if non-zero then discard the branch
                        sumc    IP,X                    'Adjust IP forward according to flag
                        jmp     #DROP                   'discard flag

' ADO = BOUNDS DO - just a quick and direct way as BOUNDS is most often never used elsewhere
' ADO ( from cnt -- )
ADO                     mov     X,tos+1
                        add     tos+1,tos
                        mov     tos,X
' DO ( to from -- )
DO                      call    #_PUSHLP                ' PUSH index onto loop stack
 ' FOR ( count -- ) Setup FOR...NEXT loop for count
FOR                     call    #_PUSHBRANCH            ' Push the IP onto the mark stack and set branch
' >L ( n -- ) Push n onto the loop stack
PUSHL                   call    #_PUSHLP
                        jmp     unext

' L> ( -- n ) Pop n from the loop stack
LPOP                    call    #LPOPX                  ' Pop loop stack into X
                        jmp     #PUSHX                  ' Push X onto the data stack as tos

' (+loop) ( n1 -- ) adds n1 to the loop index and branches back if not equal to the loop limit
PLOOP                   jmpret  POPX_ret,pDELTA wc      ' DELTA calls POPX
' The comparison above is between the call insn (wr) at DELTA and the jump insn (nr) at POPX_ret,
' this will alwaye be carry set. The call itself is indirect.

' 400ns execution time including bytecode read and execute
LOOP            if_nc   mov     X,#1                    ' default loop increment of 1
                        add     loopstk+1,X             ' increment index
                        cmps    loopstk,loopstk+1 wz,wc
BRANCH          if_a    mov     IP,branchstk            ' Branch to the address that is saved in branch
                if_a    jmp     unext
                        jmpret  LPOPX_ret,forNEXT+1 wc  ' discard top of loop index stack
                                                        ' then next loop and its branch address

' The call above borrows an indirect jump target from the call #LPOPX following the djnz at forNEXT.
' IOW it's equivalent to a jmpret LPOPX_ret, #LPOPX or call #LPOPX (ignoring flag update).

' Average execution time = 400ns
' NEXT ( -- ) Decrement count (on loop stack) and loop until 0, then pop loop stack
forNEXT         if_nc   djnz    loopstk,#BRANCH wc,wz   ' not done yet, jump to branch
                        call    #LPOPX
' POPBRANCH - pops the branch stack manually (use if forcing an exit from a stacked branch)
POPBRANCH               call                            #_POPBRANCH
                        jmp     unext



' >R ( n -- ) Push n onto the return stack
PUSHR                   mov     R0,tos
                        call    #_PUSHR
                        jmp     #DROP
' R> ( -- n ) Pop n from the return stack
RPOP                    call    #RPOPX          ' Pop return stack into R and X
                        jmp     #PUSHX          ' Push X onto the data stack as tos


' Registers can be used just like variables and the interpreted kernel uses some for itself
' 128+ bytes are reserved which can be accessed as bytes/words/longs depending upon
'  the alignment. Since the registers are pointed to by "regptr" they can relocated


' (REG) ( -- addr ) Read the next inline byte and return with the register byte address
REG                     call    #GETBYTE
                        call    #_PUSHX
' REG ( index -- addr ) Find the address of the register
ATREG                   add     tos,regptr
                        jmp     unext
```

```
' temporary timing instructions
' Capture the current cnt value and calulate cycles since last LAP - result in lapcnt
' LAP ( --  ) delta in lapcnt (created independant regs rather than REG3,4)
LAP                             neg     lapcnt,target           '    -old
                                mov     target,cnt              ' new
                                add     lapcnt,target           ' new-old
                                jmp     unext


' ABS ( n -- abs )
_ABS                            abs     tos,tos
                                jmp     unext


' um* ( u1 u2 -- u1*u2L u1*u2H ) \ unsigned 32bit * 32bit -- 64bit result - 1..11.8us
UMMUL                           mov     R0,tos+1
                        min     R0,tos                          ' max(tos, tos+1)
                        mov     R2,tos+1
                        max     R2,tos                          ' min(tos, tos+1)

                                mov     R1,#0
                                mov     tos,#0                  ' zero result
                                mov     tos+1,#0
UMMULLP                         shr     R2,#1 wz,wc             ' test next bit of u1
                        if_nc   jmp     #UMMUL1                 ' skip if no bit here
                                add     tos+1,R0 wc             ' add in shifted u2
                                addx    tos,R1                  ' carry into upper long
UMMUL1                          add     R0,R0 wc                ' shift u2 left
                                addx    R1,R1                   ' carry into 64-bits
                        if_nz   jmp     #UMMULLP                ' exhausted u1?
                                jmp     unext


' some internal code is added here so that all the XOP code is definitely in the 2nd page

                                                        { *** LITERALS *** }


' Accumulate a literal byte by byte from 1 to 4 bytes long depending upon the number of times this routine is called.
' This allows literals to be byte aligned.
ACCBYTE                 call    #GETBYTE                ' Build a big endian literal by reading in another byte
                        shl     ACC,#8                  ' merge it into the "accumulator" byte by byte
                        or      ACC,X
ACCBYTE_ret             ret

' Read the next byte of code memory via IP
GETBYTE                 rdbyte  X,IP                    ' Simply read a byte (non-code) and advance the IP
                        add     IP,#1
GETBYTE_ret             ret


' Called both by U/ and U/MOD
_UDIVMOD
                        mov     R1,#32                  ' 32 bits
                        mov     R0,#0                   ' quotient
udivmodlp               shl     tos+1, #1 wc            ' dividend
                        rcl     R0, #1                  ' hi bit from dividend
                        cmpsub  R0, tos   wc,wr         ' cmp divisor
                        rcl     R2, #1                  ' R2 - quotient
                        djnz    R1, #udivmodlp
                        mov     tos+1, R0               ' update stack with result
                        mov     tos, R2
_UDIVMOD_ret            ret

                                {{********************************** INTERNAL COG ROUTINES ********************************
                                Code from here after cog address $0FF cannot be indexed directly except by an XOP. }}


' XOP
' CMPSTR ( src dict 0 -- src dict flg ) Compare strings at these two addresses
pCMPSTR

' CMPSTR ( src dict flg-- src dict+ flg ) Compare strings at these two addresses
                                mov     R2,tos+1 wc             ' force nc on entry, s[31] == 0 (doNEXT clears c)
                                mov     X,tos+2                 ' X = source
cmpstrlp                        rdbyte  R0,X                    ' read in a character from the source
                                add     X,#1                    ' hub has to wait anyway so get ready for next source byte
                        if_c    add     R2,#1                   ' updates the copy of the dictionary pointer
                                rdbyte  R1,R2                   ' read in from the dictionary
                                cmp     R1,R0 wz                ' are they the same?
                        if_z    jmpret  par,#cmpstrlp wc,nr     ' keep at it, set carry to enable dict+ (for local copy)
nomatch                         cmp     R0,#1 wc                ' was the src null terminated? (C means we have matched the string)
                        if_c    test    R1,#$80 wc              ' set c flag if dict 8th bit set (C = cross-matched)
                                jmp     #SETZ                   ' Change our flag to -1 if C set (matched)


' used in: U@ U! STRINGs
' pRCMOVE bytes from source to destination primitive - <CMOVE conditions the parameters before calling
' XOP (RCMOVE) ( src dst cnt -- ) Copy bytes from src to dst for cnt bytes starting from the ends (in reverse)
pRCMOVE                         test    $,#1  wc                ' set carry for decrementing (always cleared by doNEXT)
```

```
' XOP (CMOVE) ( src dst cnt -- ) Copy cnt bytes from src to dst address
pCMOVE                  rdbyte  R0,bsrc                  ' read source byte
                        sumc    bsrc,#1                 ' inc or dec depending upon carry
                        wrbyte  R0,bdst                 ' write destination byte
                        sumc    bdst,#1                 ' inc or dec depending upon carry!!
                        djnz    bcnt,#pCMOVE
                        jmp     #DROP3


_COGID                  cogid   X
                        jmp     #PUSHX


' _COGINIT ( dest -- cog )
_COGINIT                coginit tos wr
                        jmp     unext

' INIT STACKS
pInitRP
                        movs    rpopins,#retstk
                        movd    _PUSHR,#retstk
                        jmp     unext



                                                { *** CONSOLE SERIAL OUTPUT *** }
{
Transmit the character that is in the tos register. This character is preconditioned with start and stop bits and the output direction is also set (to save cog space)
}
' (EMIT) ( bits -- )
pEMIT
                        and     tos,#$0FF               ' data controls loop so trim to 8-bits
                        add     tos,#$100 wc            ' add a stop bit (carry = start bit)
                        or      dira,txmask             ' make sure it's an output
                        mov     R0,cnt
                        add     R0,txticks
txbit                   muxc    outa,txmask             ' output bit
                        shr     tos,#1 wz,wc            ' lsb first
                        waitcnt R0,txticks              ' bit timing
                if_nz_or_c jmp  #txbit                  ' next bit (stop bit: z & c)
'''                     andn    dira,txmask             ' leave it to be pulled up so another cog can also use it
                        jmp     #DROP



                                                { *** PASM MODULE LOADER *** }


{
> 16 longs are reserved for a selectable module as a helper for specialized functions such as SPI etc.
LOADMOD loads the longs into this area to be executed with RUNMOD
}
' LOADMOD ( src dst cnt -- ) Load a PASM module of up to 18 bytes (or loadsz) into the cog
pLOADMOD
                        movd    lcdst,tos+1             ' Init dst pointer - just loading a small module alongside Tachyon
ldmlp                   add     lcdst,dst1              ' post-increment long destination (pipelined after rdlong)
lcdst                   rdlong  0_0,tos+2               ' read a long from hub ram into cog's destination
                        add     tos+2,#4                ' increment hub memory long address
                        djnz    tos,#ldmlp
                        jmp     #DROP3


' DELTA ( delta -- )     Calculate and set the cnt delta and waitcnt
pDELTA                  call    #POPX
                        mov     deltaR,X                ' use otherwise unused video reg for delta
                        mov     cnt,X                   ' (kernel isn't doing any video)
                        add     cnt,cnt
' WAITCNT ( -- )
pWAITCNTS               waitcnt cnt,deltaR              ' continue from last count (must be called before target is reached)
                        jmp     unext


                                                  { *** COG ACCESS *** }


' SPR@ ( index -- long ) Fetch a cog's SPR
' pSPRFETCH              add     tos,#$01F0
' COG@ ( addr -- long ) Fetch a long from cog memory
pCOGFETCH               movs    _readsrc,tos
                        nop
_readsrc                mov     tos,0_0
                        jmp     unext

' COGREG ( ix -- addr ) return with the address of the indexed cog register
pCOGREG                 add     tos,#REG0
                        jmp     unext


' COG! ( long addr -- ) Store a long to cog memory
pCOGSTORE               movd    _writedst,tos
                        nop
_writedst               mov     0_0,tos+1
                        jmp     #DROP2
```

```
' Usage: n LSTACK COG@
' Loop control words such as J K LEAVE etc implemented
' LSTACK ( index -- cog_addr ) push address of the loop stack in cog memory
pLSTACK                 add     tos,#loopstk
                        jmp     unext

{ PASM could be deprecated with the use of OPCODE instead }

' PASM ( val pasm -- result pasm )       120919 modified to not drop
pPASM                   mov     pasmins,tos
                        nop                                     ' takes care of pipeline and uses high mem for remainder of code
pasmins                 nop
                        jmp     unext

' programmable OPCODE or default fast crop operation - set mask in COGREG3 once and just issue CROP
' change at Forth level with <opcode> ' MYOP COG! etc
' NOTE: MYOP is not used in the kernel or EXTEND so it can be set by the user to suit
' MYOP ( n -- n&mask )
pMYOP
                        and     tos,#$1FF               ' setup as BLKSIZ 1- AND SDBUF + used in XADR (just for testing)
                        add     tos,sdbuf
                        jmp     unext
sdbuf                   long    @RESET+s



                                                        { *** INTERNAL STACKS *** }
{
As well as the data and return stack, a loop and branch stack is also employed
The return stack should normally only used for return addresses
A separate loop stack means that loop indicies can still be accessed in a called function
The branch stack speeds up loops by having the current loop address available without having to read an offset and calculate
The data stack is accessed as 4 fixed registers with an non-addressed overflow stack in hub RAM.
}



                                                        { *** BRANCH STACK HANDLER *** }

{
The branch stack is used for fast loop branching and so holds the branch address
It is constructed as a fixed top-of-stack location which physically moves data when it's pushed or popped
This means also that it is not possible to corrupt other memory by over or underflow.
}
_PUSHBRANCH
                        mov     branchstk+5,branchstk+4
                        mov     branchstk+4,branchstk+3
                        mov     branchstk+3,branchstk+2
                        mov     branchstk+2,branchstk+1
                        mov     branchstk+1,branchstk
                        mov     branchstk,IP            ' this is the address to loop to
_PUSHBRANCH_ret         ret

_POPBRANCH                      mov     branchstk,branchstk+1
                        mov     branchstk+1,branchstk+2
                        mov     branchstk+2,branchstk+3
                        mov     branchstk+3,branchstk+4
                        mov     branchstk+4,branchstk+5
_POPBRANCH_ret          ret


                                                        { *** LOOP STACK HANDLER *** }
{
The loop stack holds the loop limit and current index. It is constructed as a fixed top-of-stack location which physically moves data when it's pushed or popped
This means also that it is not possible to corrupt other memory by over or underflow.
}
' pop the loop stack into X
LPOPX
                                mov     X,loopstk
                                mov     loopstk,loopstk+1
                                mov     loopstk+1,loopstk+2
                                mov     loopstk+2,loopstk+3
                                mov     loopstk+3,loopstk+4
                                mov     loopstk+4,loopstk+5
                                mov     loopstk+5,loopstk+6
                                mov     loopstk+6,loopstk+7
                                mov     loopstk+7,#0
LPOPX_ret               ret

' Push tos onto the loop stack and drop tos
_PUSHLP
                                mov     loopstk+7,loopstk+6
                                mov     loopstk+6,loopstk+5
                                mov     loopstk+5,loopstk+4
                                mov     loopstk+4,loopstk+3
                                mov     loopstk+3,loopstk+2
                                mov     loopstk+2,loopstk+1
                                mov     loopstk+1,loopstk
```

```
                        mov     loopstk,tos
'
' falls through into a DROP via POPX to remove the tos
' ----> to POPX

                                                { *** DATA STACK HANDLER *** }

' Pop the data stack using fixed size stack in COG memory (allows fast direct access for operations)
' V2.2 adds an overflow stack in hub ram and reduces the size of the cog stack to 4
'
POPX                    mov     X,tos                   ' pop old tos into X
                        mov     tos,tos+1
                        mov     tos+1,tos+2
                        mov     tos+2,tos+3
                        tjz     depth,POPX_ret          ' do not allow ext stack to pop past bottom
                        sub     depth,#4
                        testn   depth,#%1111 wz         ' nz: 16+
                if_z    jmp     POPX_ret                ' direct return
                        mov     R0,stkptr
                        add     R0,depth
                        rdlong  tos+3,R0                ' pop from hub into bottom of cog stack
POPX_ret
_PUSHLP_ret
                        ret

_PUSHACC                mov     X,ACC                   ' Accumulator operation used for fast constants
_PUSHX                  mov     ACC,#0                  ' clear it for next operation
                        cmp     depth,#16 wc            ' faster stacking if we can avoid hub access, only on overflow
'                       tjz     stkptr,#PUSHCOG         ' skip external stack if no pointer assigned (set to ROM if not used)
                        mov     R0,stkptr
                        add     R0,depth
                if_nc   wrlong  tos+3,R0                ' save bottom of stack to hub RAM (only if necessary)
PUSHCOG                 mov     tos+3,tos+2             ' push the cog stack registers (4)
                        mov     tos+2,tos+1
                        mov     tos+1,tos
                        mov     tos,X                   ' replace tos with X (DEFAULT)
                        add     depth,#4                ' the depth variable indexes bytes in hub RAM (real depth = depth/4)
_PUSHACC_ret
_PUSHX_ret
                        ret



                                                { *** RETURN STACK HANDLER *** }

{
Return stack builds up in cog memory
Return stack items do not need to be directly addressed
This indexed method does not use movd and movs methods but directly inc/decs the
source and destination fields of the instruction.(retstk) so it must stay synchronized - Use !RP if needed
}
SETUPIP                 call    #GETBYTE                ' read the next byte into X and save the current IP
SAVEIP                  mov     R0,IP
_PUSHR                  mov     retstk,R0               ' save it on the stack (dest modified)
                        add     rpopins,#1              ' update source for popping
                        add     _PUSHR,dst1             ' update dest for pushing (points to next free)
SETUPIP_ret             '
SAVEIP_ret              '
_PUSHR_ret              ret



RPOPX                   sub     rpopins,#1              ' decrement rpop's source field
                        sub     _PUSHR,dst1
rpopins                 mov     X,retstk
RPOPX_ret               ret



dst1                    long    $200                    ' instruction's destination field increment
zcalls                  long    $0400-2                 ' 1K offset after XCALLs for ZCALL table

                                                { *** COG VARIABLES *** }

clockpins       long 0                                  ' I/O mask for CLOCK instruction
spisck          long 0                                  ' I/O mask for SPI clock
spiout          long 0                                  ' I/O mask for SPI data out (MOSI)
spiinp          long 0                                  ' I/O mask for SPI data in (MISO)

spice           long 0                                  ' I/O mask for SPI CE (not really required unless we use CE instr)
spicnt          long 0                                  ' bit count for variable size Kernel SPI
{
sck             res 1
mosi            res 1
miso            res 1
scnt            res 1
scs
}
```

```
' Registers used by PASM modules to hold parameters such as I/O masks and bit counts etc
REG0                    long 0
REG1                    long 0
REG2                    long 0
REG3                    long 0
REG4                    long 0

txticks                 long (sysfreq / baud )    ' set transmit baud rate
txmask                  long |<txd                ' change mask to reassign transmit
' COGREG 7 = TASK REGISTER POINTER
regptr                  long @registers+s                                   ' used by REG
Xptr                    long @XCALLS+s                                      ' used by XCALL, points to vector table (bytecode>address)
unext           long doNEXT                     ' could redirect code if used
' COGREG 10
' rearranged these register to follow REG0 so that they can be directly accessed by COGREG instruction
IP                      long 0          ' Instruction pointer
ACC                     long 0          ' Accumulator for inline byte-aligned literals
X                       long 0          ' primary internal working registers
R0                      long 0
R1                      long 0
R2                      long 0
' COGREG 16
stkptr                  long $8000              ' points to start of stack in hub ram - builds up (safe init to rom)
depth                   long 0                                              ' depth long index - points to top of overflow in hub ram
lapcnt                  long 0
target                  long 0


' *** STACKS ***
tos
datastk                 long 0[datsz]
retstk                  long 0[retsz]
loopstk                 long 0[loopsz]

branchstk               long 0[branchsz]


_RUNMOD_ ' this is a dummy symbol but the org must be equal to _RUNMOD (or less)
                        long 0[loadsz]

                        fit 496
                        long 0[$200-$]          'The kernel image becomes a general-purpose buffer and this expands it to at least 2K for coginits

' define some constants used by this cog
' The RUNMOD parameters are defined here so that the method can be changed easily

                        org tos
sdat                    res 1
            org tos
bcnt          res 1
bdst          res 1
bsrc          res 1



                        org REG0
sck                     res 1
mosi                    res 1
miso                    res 1
scs                     res 1
scnt                    res 1

                        org REG0
                        res 3
pixshift                res 1
pixeladr                res 1

endofkernel             res 0           ' just a branch to identify the end of the kernel in the listing (BST)
                        res 0
                        res 0
                        res 0
                        res 0



' If hub ram is selected for return stacks then the space here down is used. Each cog has 32 longs.
retstks

CON

instr           = $1F5
repcnt          = $1F7
deltaR          = $1FF

DAT

                                                                        { *** SERIAL RECEIVE *** }

'************************************* SERIAL RECEIVE ****************************************

{ This is a dedicated serial receive routine that runs in it's own cog }

DAT
```

```
                long 0[2]       ' read and write                        ' this hub space is used for rxwr & rxrd at runtime
rxbuffers                                                               ' hub ram gets reused as the receive buffer
                        org

HSSerialRx              mov     rxwr,#0                                 ' init write index
                        wrword  rxwr,hubrxwr                            ' clear rxrd in hub
                        wrword  rxwr,hubrxrd                            ' make rxwr = rxrd
                        mov     stticks,rxticks                         ' calculate start bit timing
                        shr     stticks,#1                              ' half a bit time less
                        sub     stticks,#4                              ' compensate timing - important at high speeds
                        mov     breakcnt,#200                           ' reset break count

'-------------------------------------------------------------------------------
receive                 mov     rxcnt,stticks                           ' Adjusted bit timing for start bit
                        waitpne rxpin,rxpin                             ' wait for a low = start bit
'
' START BIT DETECTED
'                               ' time sample for middle of start bit
                        add     rxcnt,cnt                               ' uses special start bit timing
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina       wz                      ' sample middle of start bit
rxcond2         if_nz   jmp     #receive                                ' restart if false start otherwise bit time from center

'
' START bit validated
' Read in data bits lsb first
' No point in looping as we have plenty of code to play with
' and inlining (don't call) and unrolling (don't loop) can lead to higher receive speeds
'
                        waitcnt rxcnt,rxticks                           ' wait until middle of first data bit
                        test    rxpin,ina wc                            ' sample bit 0
                        muxc    rxdata,#01                              ' and assemble rxdata
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina wc                            ' sample bit 1
                        muxc    rxdata,#02
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina wc                            ' sample bit 2
                        muxc    rxdata,#04
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina wc                            ' sample bit 3
                        muxc    rxdata,#08
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina wc                            ' sample bit 4
                        muxc    rxdata,#$10
 ' data bit 5
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina wc                            ' sample bit 5
                        muxc    rxdata,#$20
                        mov     X1,rxbuf                                ' squeeze in some overheads, calc write pointer
                        add     X1,rxwr                                 ' X points to buffer location to store
' data bit 6
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina wc                            ' sample bit 6
                        muxc    rxdata,#$40
                        add     rxwr,#1                                 ' update write index
                        and     rxwr,wrapmask
' last data bit 7
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina wc                            ' sample bit 7
                        muxc    rxdata,#$80
                        wrbyte  rxdata,X1                               ' save data in buffer - could take 287.5ns worst case
' stop bit
stopins                 waitcnt rxcnt,rxticks                           ' check stop bit early (need to detect errors and breaks)
                        test    rxpin,ina wc                            ' sample stop bit
                if_nc   jmp     #frmerror                               ' framing error - check for break - disregard timing now
{
'
' MULTIDROP
'
                        mov     rxchk,rxdata
                        andn    rxchk,#$0F
                        cmp     rxchk,#$F0 wz                           ' if $Fx address detected then firstly disable rx pass
                if_z    mov     rxon,#0
                        cmp     rxdata,#$FF wz
                if_z    mov     rxon,rxdata
                        cmp     rxdata,myadr wz
                if_z    mov     rxon,myadr                              ' indicate that it is only this unit responding
}
                if_c    wrword  rxwr,hubrxwr                            ' update hub index for code reading the buffer if all good
                        wrbyte  rxdata,lkptr
                        jmp     #receive

' Framing error - check if it's a null character as it may be part of a break condition
'
frmerror
                        sub     rxwr,#1
                        cmp     rxdata,#0 wz                            'if it's a null it could be part of a break
                if_nz   mov     breakcnt,#200                           'reset the break count (compromise here to keep main loop tight)
                if_nz   jmp     #receive                                'ignore normal framing error
                        or      outa,rxpin
                        or      dira,rxpin                              'unintentional? make sure the input is not floating
                        mov     rxcnt,#16
                        djnz    rxcnt,$
```

```
                        andn    dira,rxpin              'restore input and delay
                        mov     rxcnt,#16
                        djnz    rxcnt,$
                        test    rxpin,ina wz            'if it's still low then it is being intentionally driven
                if_nz   jmp     #receive                'ignore floating input
                        djnz    breakcnt,#receive
aboot
                        mov     rxcnt,#$80
                        clkset  rxcnt                   ' reboot


-------------------------------------------------------------------------------

lkptr                   long    @registers+s+lastkey    ' receive cog writes directly to lastkey register
rxpin                   long    |<rxd                    ' mask of rx pin
hubrxrd                 long    @rxbuffers+s-4          ' ptr to rxrdin hub
hubrxwr                 long    @rxbuffers+s-2          ' word address of rxwr in hub (after init)
rxbuf                   long    @rxbuffers+s            ' pointer to rxbuf in hub memory
breakcnt                long    40
wrapmask                long    rxsize-1
rxon                    long    0
mode                    long    0
rxticks                 long    0
stticks                 long    0
spticks                 long    0
rxcnt                   long    0
rxdata                  long    0                       'assembled character
lastch                  long    0
X1                      long    0
rxchk                   long    0
rxwr                    long    0                       'cog receive buffer write index - copied to hub ram
end                     res     0

-------------------------------------------------------------------------------

{ LOOP VERSION

DAT
                long 0[2]       ' read and write        ' this hub space is used for rxwr & rxrd at runtime
rxbuffers                                               ' hub ram gets reused as the receive buffer
                        org

HSSerialRx              mov     rxwr,#0                 ' init write index
                        wrword  rxwr,hubrxwr            ' clear rxrd in hub
                        wrword  rxwr,hubrxrd            ' make rxwr = rxrd
                        mov     stticks,rxticks         ' calculate start bit timing
                        shr     stticks,#1              ' half a bit time less
                        sub     stticks,#4              ' compensate timing - important at high speeds
                        mov     breakcnt,#200           ' reset break count

-------------------------------------------------------------------------------

receive                 mov     rxcnt,stticks           ' Adjusted bit timing for start bit
                        waitpne rxpin,rxpin             ' wait for a low = start bit
'
' START BIT DETECTED
'                               ' time sample for middle of start bit
                        add     rxcnt,cnt               ' uses special start bit timing
                        waitcnt rxcnt,rxticks
                        test    rxpin,ina       wz      ' sample middle of start bit
rxcond2         if_nz   jmp     #receive                ' restart if false start otherwise bit time from center
'
' START bit validated
' Read in data bits lsb first
                        mov     R1,#9                   ' for 8 data bits + stop bit
rxloop                  waitcnt rxcnt,rxticks           ' wait until middle of first data bit
                        test    rxpin,ina wc            ' sample bit 0
                        rcr     rxdata,#1
                        djnz    R1,#rxloop
                if_nc   jmp     #frmerror               ' framing error - check for break - disregard timing now
                        mov     X,rxbuf
                        add     X,rxwr
                        add     rxwr,#1                 ' update write index
                        and     rxwr,wrapmask
                        wrbyte  rxdata,X                ' save data in buffer - could take 287.5ns worst case
                        wrword  rxwr,hubrxwr            ' update hub index for code reading the buffer if all good
                        wrbyte  rxdata,keyptr           ' update lastkey variable for Forth to check
                        jmp     #receive

' Framing error - check if it's a null character as it may be part of a break condition
'
frmerror
                        sub     rxwr,#1
                        cmp     rxdata,#0 wz            'if it's a null it could be part of a break
                if_nz   mov     breakcnt,#200           'reset the break count (compromise here to keep main loop tight)
                if_nz   jmp     #receive                'ignore normal framing error
                        or      outa,rxpin
                        or      dira,rxpin              'unintentional? make sure the input is not floating
                        mov     rxcnt,#16
                        djnz    rxcnt,$
                        andn    dira,rxpin              'restore input and delay
```

```
                        mov     rxcnt,#16
                        djnz    rxcnt,$
                        test    rxpin,ina wz            'if it's still low then it is being intentionally driven
                if_nz   jmp     #receive                'ignore floating input
                        djnz    breakcnt,#receive
aboot
                        mov     rxcnt,#$80
                        clkset  rxcnt                   ' reboot


---

keyptr                  long    @registers+s+lastkey    ' receive cog writes directly to lastkey register
rxpin                   long    |<rxd                   ' mask of rx pin
hubrxrd                 long    @rxbuffers+s-4          ' ptr to rxrdin hub
hubrxwr                 long    @rxbuffers+s-2          ' word address of rxwr in hub (after init)
rxbuf                   long    @rxbuffers+s            ' pointer to rxbuf in hub memory
breakcnt                long    40
wrapmask                long    rxsize-1
mode                    long    0
rxticks                 long    0
stticks                 long    0
spticks                 long    0
rxcnt                   long    0
rxdata                  long    0                       'assembled character
lastch                  long    0
X1                      long    0
savdat                  long    0
rxwr                    long    0                       'cog receive buffer write index - copied to hub ram
end                     res     0

}


{ Boot-time Spin startup - launches Tachyon into all the remaining cogs and starts serial receive in this cog = 0 }

PUB Start
  rxticks := txticks := clkfreq / baudrate   ' Force VM transmit routine to correct baud
  coginit(1,@HSSerialRx, @rxbuffers)
  repeat 6
                cognew(@RESET, @IDLE_reset)
  coginit(0,@RESET, @TERMINAL_reset)



{ Extend this kernel by pasting or sending EXTEND.fth to the Prop running the kernel - use 12ms line delay or more }


'( LINK TO EXTEND.fth )
```