PropForth Core(\) and Optional(\\) Words - Quick Reference

****

TRUE Condition NOT= 0, any value negative or positive
FALSE Condition = 0

High Pin Condition = 1
Low Pin Condition = 0

Arithmetic and Logical Operations

```
\   and ( n1 n2 -- n1 )             bitwise and n1 n2
\   andn ( n1 n2 -- n1 )            bitwise and n1 invert n2
\   or ( n1 n2 -- n1_or_n2 )        bitwise or
\   xor ( n1 n2 -- n1_xor_n2 )      bitwise xor
\   invert ( n1 -- n2 )             bitwise invert n1
\   lshift (n1 n2 -- n3)            n3 = n1 shifted left n2 bits
\   rshift ( n1 n2 -- n3)           n3 = n1 shifted right logically n2 bits
\   rashift ( n1 n2 -- n3)          n3 = n1 shifted right arithmetically n2 bits
\\  andC! ( c1 addr -- )            and c1 with the contents of address
\   W+! ( n1 addr -- )              add n1 to the word contents of address
\   orC! ( c1 addr -- )             or c1 with the contents of address
\   andnC! ( c1 addr -- )           and inverse of c1 with the contents of address
\   orlnfa ( c1 -- )                ors c1 with the nfa length of the last name field
                                          entered

\   between ( n1 n2 n3 -- t/f )     true if n2 <= n1 <= n3

\   negate ( n1 -- 0-n1 )           the negative of n1
\   max ( n1 n2 -- n1 )             signed max of top 2 stack values
\   min ( n1 n2 -- n1 )             signed min of top 2 stack values
\\  abs ( n1 -- abs_n1 )            absolute value of n1
\\  sign ( n1 n2 -- n3 )            n3 is the xor of the sign bits of n1 and n2

\   - ( n1 n2 -- n1-n2 )            difference of ni & n2
\   + ( n1 n2 -- n1+n2 )            sum of n1 & n2
\\  * ( n1 n2 -- n1*n2)             n1 multiplied by n2
\\  / ( n1 n2 -- n1/n2)             n1 divided by n2
\\  */mod ( n1 n2 n3 -- n4 n5 )     n5 = (n1*n2)/n3, n4 is the remainder.
                                          Uses a 64bit intermediate result.
\\  */ ( n1 n2 n3 -- n4 )           n4 = (n1*n2)/n3. Uses a 64bit intermediate result.
\\  /mod ( n1 n2 -- n3 n4 ) \       signed divide & mod  n4 = n1/n2, n3 is the
                                          remainder

\   u/ ( u1 u2 -- u1/u2)            u1 divided by u2
\   u* ( u1 u2 -- u1*u2)            u1 multiplied by u2
\   um* ( u1 u2 -- u1*u2L u1*u2H )  unsigned 32bit * 32bit -- 64bit result
\   um/mod ( u1lo u1hi u2 -- remainder quotient )
                                    unsigned divide & mod  u1 divided by u2
\   u/mod ( u1 u2 -- remainder quotient )
                                    unsigned divide & mod  u1 divided by u2
\\  u*/mod ( u1 u2 u3 -- u4 u5 )    u5 = (u1*u2)/u3, u4 is the remainder. Uses a 64bit
                                          intermediate result.
\\  u*/ ( u1 u2 u3 -- u4 )          u4 = (u1*u2)/u3 Uses a 64bit intermediate result.

\   1+ ( n1 -- n1+1 )               add 1
\   1- ( n1 -- n1-1 )               subtract 1
\   2+ ( n1 -- n1+2 )               add 2
\   2- ( n1 -- n1-2 )               subtract 2
\   4+ ( n1 -- n1+4 )               add 4
\   2* ( n1 -- n1<<1 )              n2 is shifted logically left 1 bit
\   4* ( n1 -- n1<<1 )              n2 is shifted logically left 2 bits
```

```
\   2/ ( n1 -- n1>>1 )              n2 is shifted arithmetically right 1 bit
\\  rnd ( -- n1 )                  n1 is a random number from 00 - FF
\\  rndtf ( -- t/f)                 true or false randomly
\\  rndand ( n1 -- n2)              n2 is randomly n1 or 0
```

Number Type Conversions

```
\   w>l ( n1 n2 -- n1n2 )           consider only lower 16 bits
\   l>w ( n1n2 -- n1 n2)            break into 16 bits
\   tochar ( n1 -- c1 )             convert n1 to a char
```

Comparison Operators

```
\   = ( n1 n2 -- t/f )              compare top 2 32 bit stack values,
                                          true if they are equal
\   > ( n1 n2 -- t/f )              flag is true if and only if n1 is greater than n2
\   < ( n1 n2 -- t/f )              flag is true if and only if n1 is less than n2
\   <> ( x1 x2 -- flag )            flag is true if and only if x1 is not bit-for-bit
                                          the same as x2
\   >= ( n1 n2 -- t/f)              true if n1 >= n2
\   <= ( n1 n2 -- t/f)              true if n1 <= n2
\   0= ( n1 -- t/f )                true if n1 is zero
\   0<> ( n1 -- t/f )               true if n1 is not zero
\   0< ( n1 -- t/f )                true if n1 < 0
\   0> ( n1 -- t/f )                true if n1 > 0
\   0>= ( n1 -- t/f )               true if n1 >= 0
```

Manipulating the Stack

```
\   drop ( n1 -- )                  drop the value on the top of the stack
\   dup ( n1 -- n1 n1 )
\   over ( n1 n2 -- n1 n2 n1 )      duplicate 2 value down on the stack to the
                                          top of the stack
\   rot ( n1 n2 n3 -- n2 n3 n1 )    rotate top 3 value on the stack
\   rot2 ( x1 x2 x3 -- x3 x1 x2 )
\   swap ( n1 n2 -- n2 n1 )         swap top 2 stack values
\   2dup ( n1 n2 -- n1 n2 n1 n2     copy top 2 items on the stack
\   2drop ( n1 n2 -- )              drop top 2 items on the stack
\   3drop ( n1 n2 n3 -- )           drop top 3 items on the stack
\   nip ( x1 x2 -- x2 )             delete the item x1 from the stack
\   tuck ( x1 x2 -- x2 x1 x2        copy top item into 3rd stack slolt

\   r> ( -- n1 )                    pop top of RS to stack
\   >r ( n1 -- )                    pop stack top to RS
\   2>r ( n1 n2 -- )                pop top 2 stack top to RS
\\  r@ ( -- n1 ) \                  copy top of RS to stack

\   bounds ( x n -- x+n x )
```

Memory-Stack Transfer

```
\   L@ ( addr -- n1 )              fetch 32 bit value at main memory addr
\   C@ ( addr -- c1 )              fetch 8 bit value at main memory addr
\   W@ ( addr -- h1 )              fetch 16 bit value at main memory addr

\   L! ( n1 addr -- )              store 32 bit value (n1) at main memory addr
\   C! ( c1 addr -- )              store 8 bit value (c1) main memory at addr
\   W! ( h1 addr -- )              store 16 bit value (h1) main memory at addr

\   litw ( -- h1 )                 push a 16 bit literal on the stack
\   litl ( -- n1 )                 push a 32 bit literal on the stack

\   C@++ ( c-addr -- c-addr+1 c1 )  fetch the character and increment the address
```

Definite Loops

```
\   do     ( n1 -- )               marks start of a block run 1 or more times
(immediate)
\   doloop                         used in loop & +loop
\   (loop) ( -- )                  add 1 to loop counter, branch if count is below
                                       limit,offset follows
\   loop                           marks end of do block (immediate)
\   (+loop) ( n1 -- )              add n1 to loop counter, branch if count is below
                                       limit,offset follows
\   +loop                          marks end of do block (immediate)
\   leave ( -- )                   exits at the next loop or +loop, i is placed to the
                                       max loop value
\   i ( -- n1 )                    the most current loop counter
\   ibound ( -- n1 )               the upper bound of i
\\  lasti? ( -- t/f )              true if this is the last value of i in this loop
\   seti ( n1 -- )                 set the most current loop counter

\\  j ( -- n1 )                    the second most current loop counter
\\  jbound ( -- n1 )               the upper bound of j
\\  lastj? ( -- t/f )              true if this is the last value of j in this loop
\\  setj ( n1 -- )                 set the second most current loop counter
```

Indefinite Loops

```
\   begin  ( -- )                  marks beginning of a group of words to be executed
\   until  ( t/f -- )              iterates back to begin until true
```

String Operators

```
\   <# ( -- )                      initialize the output area
\   #> ( -- caddr )                address of a counted string representing the
                                       output, NOT ANSI
\   # ( n1 -- n2 )                 divide n1 by base and convert the remainder to a
                                       char and append to the output
\   #s ( n1 -- 0 )                 execute # until the remainder is 0
```

```
\   cmove ( c-addr1 c-addr2 u -- )  If u is greater than zero, copy u consecutive
                                     characters from the data space starting
                                     at c-addr1 to that starting at c-addr2, proceeding
                                     character-by-character from lower addresses to
                                     higher addresses.

\   ctoupper ( c1 -- c1 )           if c is a-z converts it to upper case
\   todigit ( c1 -- n1 )            converts character to a number
\   isdigit ( c1 -- t/f )           true if is it a valid digit according to base
\   isunumber ( c-addr len -- t/f ) true if the string is numeric
\   unumber ( c-addr len -- u1 )    convert string to an unsigned number
\   number ( c-addr len -- n1 )     convert string to a signed number
\   isnumber ( c-addr len -- t/f )  true if the string is numeric

\   fill ( c-addr u char -- )       set string at addr to u characters
\   accept ( c-addr +n1 -- +n2 )    collect n1 -2 characters or until eol,
                                          convert tab to space,
\   find ( c-addr -- c-addr 0 | xt 2 | xt 1  |  xt -1 )
                                    c-addr is a counted string,
                                          0 - not found, 2 eXecute word,
                                          1 immediate word, -1 word NOT ANSI
```

Input/Output Operations

```
\   cr ( -- )                       emits a carriage return
\   space ( -- )                    emits a space
\   spaces ( n -- )                 emit n spaces
\   _ecs                            emit esc charactere

\   .bvalue ( n1 -- )               emits 3 character
\   .addr ( n1 -- )                 emits 6 character address
\   .value ( n1 -- )                emits 11 character number
\   .hex ( n -- )                   emit a single hex digit
\   .byte ( n -- )                  emit 2 hex digits
\   .word ( n -- )                  emit 4 hex digits
\   .str ( c-addr u1 -- )           emit u1 characters at c-addr
\\  .long ( n -- )                  emit 8 hex digits
\   . ( n1 -- )                     displays top of stack
```

These routines below will output to the console. They will not block, so characters may drop in the case of collisions.

```
\\  .conemit ( c1 -- )              emit cr to console, will timeout, 80MHZ cog 57.6Kb
                                          console, should keep up
\\  .concstr ( cstr -- )            emit cstr to console
\\  .con ( n1 -- )                  print n1 to the console to console
\\  .concr ( -- )                   emit a cr to the console
\\  .conbvalue
\\  .conaddr
\\  .convalue
\\  .const? ( -- )                  prints out the stack

\   emit? ( -- t/f)                 true if the output is ready for a char
\   femit? (c1 -- t/f)              true if the output emitted a char, a fast non
                                          blocking emit
\   emit ( c1 -- )                  emit the char on the stack
\   key? ( -- t/f)                  true if there is a key ready for input
\   fkey? ( -- c1 t/f )             fast nonblocking key routine, true if c1 is a valid
                                          key
\   key ( -- c1 )                   get a key
```

```
\   clearkeys ( -- )                clear the input keys

\   delms ( n1 -- )                 delays n1 millisec (for 80Mhz 68DB max)
\\  cappendc ( c1 cstr -- )          append c1 the cstr
\\  cappendnc ( n cstr -- )          print the number n and append to cstr and then
                                        append a blank
\\  ctolower ( c1 -- c1 )            if c is A-Z converts it to lower case
\\  #C ( c1 -- )                     prepend the character c1 to the number currently
                                        being formatted
\\  .cogch ( n1 n2 -- )              print as x(y)

\   alignl ( n1 -- n1)              aligns n1 to a long (32 bit)  boundary
\   alignw ( n1 -- n1)              aligns n1 to a halfword (16 bit)  boundary
\   npfx ( c-addr1 c-addr2 -- t/f ) -1 if c-addr2 is prefix of c-addr1, 0 otherwise
\   namelen ( c-addr -- c-addr+1 len )
                                    returns c-addr+1 and the length of the name at
                                        c-addr
\   namecopy ( c-addr1 c-addr2 -- ) Copy the name from c-addr1 to c-addr2
\   ccopy ( c-addr1 c-addr2 -- )    Copy the cstr from c-addr1 to c-addr2
\   cappend ( c-addr1 c-addr2 -- )  addpend the cstr from c-addr1 to c-addr2
\   cappendn ( n cstr -- )          print the number n and append to cstr
\   .strname ( c-addr -- )          c-addr point to a forth name field, print the name
\   .cstr ( addr -- )               emit a counted string at addr
\   dq ( -- )                       emit a counted string at the ip, and increment the
                                        ip past it and word alignw it
```

File Input/Output (eeprom)

```
\\  ee>image ( addr n1 -- )         addr - the address to start producing an image
                                        from ,n1 - count of bytes to produce,
                                        must be a multiple of 64
\\  image>ee ( n1 - n16 addr -- )   writes the 16 longs on the stack to the addr in
                                        eeprom
```

Eeprom read and write routines below are for the Prop Proto Board AT24CL256 eeprom on pin 28 sclk, 29 sda.

The eereadpage and eewritePage words assume the eeprom is 64kx8 or larger and will address up to 8 sequential eeproms.

```
\   _eestart ( -- )                 start the data transfer
\   _eestop ( -- )                  stop the data transfer
\   _eewrite ( c1 -- t/f )          write a byte to the eeprom, returns ack bit
\   eewritepage ( eeAddr addr u -- t/f )
                                    return true if there was an error, use lock 1
\\  eereset ( -- )                  initialize the eeprom in case it is in a weird
                                        state
\\  _eeread ( t/f -- c1 )           read a byte from the eeprom, ackbit in, byte out
\\  eereadpage ( eeAddr addr u -- t/f )
                                    return true if there was an error, use lock 1
\   EW! ( n1 eeAddr -- )
\\  EW@ ( eeAddr -- n1 )
\\  EC@ ( eeAddr -- c1 )
\\  eecopy ( addr1 addr2 u -- )     copy u bytes from addr1 to addr2, addr1 and addr2
                                        must beon a 0x40 byte page boundary
                                        clears the pad, so make sure no commands
                                        follow and u must be a multiple of 0x40 and
                                        should not overlap
```

Inspect Memory, Debug & View Code

```
\   words ( -- )                    prints the words in the forth dictionary, if the
                                    pad has another string following, with that prefix
\\  st? ( -- )                       prints out the stack
\\  sc ( -- )                       clears the stack
\\  _pna ( addr -- )                print the address, contents and forth name
\\  rs? ( -- )                      prints out the return stack

\\  (forget) ( cstr -- )            wind the dictionary back to the word which follows
\\  forget ( -- )                   wind the dictionary back to the word which follows
\\  free ( -- )                     display free main bytes and current cog longs

\\  (dumpb)
\\  (dumpm)
\\  (dumpe)

\\  dump  ( adr cnt -- )            uses tbuf
\\  edump  ( adr cnt -- )           uses tbuf
\\  cogdump  ( adr cnt -- )

        Noisy reset messages print out a reset message to the console

\   (rsm) ( n -- ) n is the last status
\   0011FFFF - stack overflow
\   0012FFFF - return stack overflow
\   0021FFFF - stack underflow
\   0022FFFF - return stack underflow
\   8100FFFF - no free cogs
\   8200FFFF - no free main memory
\   8400FFFF - fl no free main memory
\   8500FFFF - no free cog memory
\   8800FFFF - eeprom write error
\   9000FFFF - eeprom read error
```

Define Data Structures

```
\   doconw ( -- h1 )                push 16 bit constant which follows on the stack
                                          - implicit a_exit
\   doconl ( -- n1 )                push a 32 bit constant which follows the stack
                                          - implicit a_exit
\   dovarw ( -- addr )              push address of 16 bit variable which follows on
                                          the stack - implicit a_exit
\   asmlabel ( x -- )               skip blanks parse the next word
                                          and create an assembler entry
\   dovarl ( -- addr )              push address of 32 bit variable which follows the
                                          stack - implicit a_exit
\   wconstant ( x -- )              skip blanks parse the next word and create a
                                          constant allocate a word, 2 bytes
\   wvariable ( -- )                skip blanks parse the next word and create a
                                          variable  allocate a word, 2 bytes
\\  cogvariable ( -- )              skip blanks parse the next word and create a cog
                                          variable, allocate a long
\\  variable ( -- )                 skip blanks parse the next word and create a
                                          variable,allocate a long, 4 bytes
\\  constant ( x -- )               skip blanks parse the next word and create a
                                          constant,allocate a long, 4 bytes
```

Propeller-Specific Words (Not listed in other categories.)

Cog related words

```
\   cogstop ( n -- )
\   cogreset ( n1 -- )              reset the forth cog
\   cogio ( n -- addr)              the address of the data area for cog n
\   cogiochan ( n1 n2 -- addr )     cog n1, channel n2 ->addr
\   io>cogchan ( addr -- n1 n2 )    addr -> n1 cogid, n2 channel
\   io>cog ( addr -- n )            addr -> cogid
\   io  ( -- addr )                 the address of the io channel for the cog
\   coghere ( -- addr )             access as a word, the first unused register address
                                        in this cog
\   cogncchan ( n1 -- n2 )          number of io channels for cog n2
\   >con ( n1 -- )                  disconnect the current cog, and connect the console
                                        to the forth cog
\   debugcmd  ( -- addr )           address of the debugcmd as a word, used to
                                        commincate from forth cog to request a reset,
                                        or for traces
\   COG@ ( addr -- n1 )             fetch 32 bit value at cog addr
\   COG! ( n1 addr -- )             store 32 bit value (n1) at cog addr
\   (nfcog) ( -- n1 n2 )            n1 the next valid free forth cog, n2 is 0 if the
                                        cog is valid
\   nfcog ( -- n )                  returns the next valid free forth cog
\   cogx ( cstr n -- )              execute cstr on cog n
\   cogid ( -- n1 )                 return id of the current cog ( 0 - 7 )
\   cogpad
\   cognumpad
\   cogstate

\   (iodis) ( n1 n2 -- )            cog n1 channel n2 disconnect, disconnect this cog
                                        and the cog it is connected to
\   iodis ( n1 -- )                 cogid to disconnect, disconnect this cog and the
                                        cog it is connected to
\   (ioconn) ( n1 n2 n3 n4 -- )     connect cog n1 channel n2 to cog n3 channel n4,
                                        disconnect them from other cogs first
\   ioconn ( n1 n2 -- )             connect the 2 cogs, disconnect them from other cogs
                                        first
\   (iolink) ( n1 n2 n3 n4 -- )     links the 2 channels, output of cog n1 channel n2
                                        -> input of cog n3 channel n4, output of n3
                                        channel n4 -> old output of n1 channel n2
\   iolink ( n1 n2 -- )             links the 2 cogs, output of n1 -> input of n2,
                                        output of n2 -> old output of n1
\   (iounlink) ( n1 n2 -- )         unlinks cog n1 channel n2
\   iounlink ( n1 -- )              unlinks the cog n1

\\  cog+ ( -- )                     add a forth cog
\\  (cog-) ( -- )                   stop first forth cog, cannot be executed from
                                        the first forth cog
\\  cog- ( -- )                     stop first forth cog, cannot be executed from
                                        the first forth cog
\\  cog? ( -- )

\\  aallot ( n1 -- )                add n1 to coghere, allocates space in the cog or
                                        release it, n1 is # of longs
\\  cog, ( x -- )                   allocate 1 long in the cog and copy x to that
                                        location
\   hubop ( n1 n2 -- n3 t/f )       n2 specifies which hubop (0 - 7), n1 is the source
                                        datcog,
```

```
\   ERR ( n1 -- )                  clear the input queue, set the error n1 and reset
                                       this cog
\   cogdebugcmd
\   debugvalue  ( -- addr )        the address of the debugvalue as a long,
                                       used in conjuction with debugcmd
\   cogdebugvalue                  n3 is returned, t/f is the 'c' flag is set from the
                                       hubop
```

Locks

```
\   lockset ( n1 -- n2 )           set lock n1, result is in n2, -1 if the lock was
                                       set as per 'c' flag, lock ( n1 ) must have
                                       been allocated via locknew
\   lockclr ( n1 -- n2 )           clear lock n1, result is in n2, -1 if the lock was
                                       set as per 'c' flag, lock ( n1 ) must have
                                       been                    allocated via locknew
\   lockdict? ( -- t/f )           attempt to lock the forth dictionary, 0 if
                                       unsuccessful -1 if successful
\   freedict ( -- )                free the forth dictionary, if I have it locked
\   lockdict ( -- )                lock the forth dictionary
\   mydictlock ( -- addr )         access as a char, the number of times dictlock has
                                       been executed in the cog minus the freedict
\\  locknew ( -- n2 )              allocate a lock, result is in n2, -1 if
                                       unsuccessful
\\  lockret ( n1 -- )              deallocate a lock, previously allocated via locknew
```

Propeller i/o pins

```
\   pinin ( n1 -- )                set pin # n1 to an input
\   pinout ( n1 -- )               set pin # n1 to an output
\   pinlo ( n1 -- )                set pin # n1 to low
\   pinhi ( n1 -- )                set pin # n1 to high
\   px ( t/f n1 -- )               set pin # n1 to h - true or l false
\\  px? ( n1 -- t/f)               true if pin n1 is hi


\   reboot ( -- )                  reboot the propellor chip
\   reset ( -- )                   reset this cog
```

Registers (wconstants)

```
\   par     \   ina    \   dira  \\  ctra  \\  frqa  \\  phsa  \\  vsfg
\   cnt     \   outa   \\  ctrb  \\  frqb  \\  phsb   \\ vscl

\   parat ( offset -- addr )       the offset is added to the contents of the par
                                       register,giving an address references
```

Clock related

```
\   clkfreq ( -- u1 )              the system clock frequency
\\  waitcnt ( n1 n2 -- n1 )        wait until n1, add n2 to n1
\\  waitpeq ( n1 n2 -- )           wait until state n1 is equal to ina anded with n2
\\  waitpne ( n1 n2 -- )           wait until state n1 is not equal to ina anded with
                                       n2

\\  _cfo ( n1 -- n2 )              n1 - desired frequency, n2 freq a
\\  setHza ( n1 n2 -- )            n1 is the pin, n2 is the freq, uses ctra set
                                       the pin oscillating at the specified
                                       frequency
\\  qHzb ( n1 n2 -- n3 )           n1 - the pin, n2 - the # of msec to sample, n3 the
                                       frequency
```

```
\\  setHzb ( n1 n2 -- )       n1 is the pin, n2 is the freq, uses ctrb
                                  set the pin oscillating at the specified
                                  frequency
```

Compiler & Interpreter System Extensions

```
\   >out ( -- addr )          access as a word, the offset to the current output byte
\   >in ( -- addr )           access as a word, addr is the var the offset in
                                  characters from the start of the input buffer to
                                  the parse area.
\   pad  ( -- addr )          access as bytes, or words and long, the address of the
                                  pad area - used by accept for keyboard input, can
                                  be used carefully  by other code
\   pad>in ( -- addr )        addr is the address to the start of the parse area.
\   namemax ( -- n1 )         the maximum name length allowed must be 1F80 wconstant
\   padsize                   the size of the pad area
\   execword ( -- addr )      a long, an area where the current word for execute is
                                  stored
\   execute ( addr -- )       execute the word - pfa address is on the stack
\   pad>out ( -- addr )       addr is the address to the the current output byte

\   numpad ( -- addr )        the of the area used by the numeric output routines,
                                  can be used carefully by other code
\   numpadsize                the size of the numpad, 0x21 bytes if we are working in
                                  binary, otherwise 1 + max num digits are necessary

\   state ( -- addr)          access as a char

              bit 0 -  0 - interpret mode  / 1 - forth compile mode
              bit 1 -  0 - direct console output turned off /
                               1 - direct console output turned on
              bit 2 -  0 - Free / 1 - PropForth cog
              bit 3 -  0 - Free / 1 - Other cog does not support IO channels
              bit 4 -  0 - Free / 1 - Other cog supports io channels
              bit 5 - 7 - number of io channels - 1

\   compile? ( -- t/f )       true if we are in a compile. Pad with 1 space at start &
                                  end.
\   parse ( c1 -- +n2 )       parse the word delimited by c1, or the end of buffer
                                  is reached, n2 is the length >in is the offset
                                  in the pad of the start of the parsed word
\   skipbl ( -- )             increment >in past blanks or until it equals padsize
\   nextword ( -- )           increment >in past current counted string
\   parseword ( c1 -- +n2 )   skip blanks, and parse the following word delimited by
                                  c1,update to be a counted string in the pad
\   parsebl ( -- t/f)         parse the next word in the pad delimited by blank,
                                  true if there is a word
\   parsenw ( -- cstr )       parse and move to the next word, str ptr is zero
                                  if there is no next word
\\  parsenw ( -- cstr )       parse and move to the next word, str ptr is zero
                                  if there is no next word
\\  padnw ( -- t/f )          move past current word and parse the next word, true
                                  if there is a next word
\\  padclr ( -- )
\\  padbl ( -- )              fills this cogs pad with blanks

\   ccreate ( cstr -- )       create a dictionary entry
\   create ( -- )             skip blanks parse the next word and create a dictionary
                                  entry
```

```
\   herewal ( -- )            align contents of here to a word boundary, 2 byte
                                    boundary
\   allot ( n1 -- )           add n1 to here, allocates space on the data dictionary
                                    or release it
\   herelal ( -- )            alignw contents of here to a long boundary, 4 byte
                                    boundary
\   immediate ( -- )          marks last entry as an immediate word
\   exec ( -- )               marks last entry as an eXecute word, executes always

\   w, ( x -- )               allocate 1 halfword 2 bytes in the dictionary
                                    and copy x to that location
\   c, ( x -- )               allocate 1 byte in the dictionary and copy x to that
                                    location
\   l, ( x -- )               allocate 1 long, 4 bytes in the dictionary
                                    and copy x to that location
\   ' ( -- addr )             returns the execution token for the next name,
                                    if not found it returns 0
```

Defining Routines

```
\   exit     ( -- )           exit the current forth word, and back to the caller
\   branch                    16 bit branch offset follows -
                                    -2 is to itself, +2 is next word
\   0branch ( t/f -- )        branch it top of stack value is zero
                                    16 bit branch offset follows

\   dothen
\   then
\   thens
\   if
\   else

:  ( -- )                     : <name> starts "colon: definition for name
;  ( -- )                     finish defining the "colon" definition

\   forthentry ( -- )         marks last entry as a forth word

\   cq ( -- addr )            returns the address of the counted string following this
                                    word and increments the IP past it
\   c" ( -- c-addr )          compiles the string delimited by ", runtime return the
                                    addr of the counted string ** valid only in that
                                    line
```

```
: _sp w, 1 >in W+! 22 parse dup c, dup pad>in here W@ rot cmove dup allot 1+ >in W+!
herewal ;

: ." $H_dq _sp ;  immediate
```

Other

Change Compilation & Interpretation Settings

```
\   base ( -- addr )          access as a word, the address of the base variable
\   hex ( -- )                set the base for hexadecimal
\   decimal ( -- )            set the base for decimal
```

Comment Introducing Operators

```
\   \   ( -- )                moves the parse pointer >in to the end of the line
```

```
\   {   ( -- )              discard all the characters between { and }

NOTE - The opening brace MUST be the first and only character on a new line, the
closing brace must be alone on another line - does not work in compile mode.
\   }

\   [if xxx   ( -- )        if xxx is defined drop all characters until ],
                            [if xxx not have any characters following it on the line
```

Word structure

```
\   nfa>lfa ( addr -- addr )  go from the nfa (name field address) to the lfa
                                  (link field address)
\   nfa>pfa ( addr -- addr )  go from the nfa (name field address) to the pfa
                                  (parameter field address)
\   nfa>next ( addr -- addr ) go from the current nfa to the prev nfa in the dictionary
\   lastnfa ( -- addr )       gets the last NFA
\   isnamechar ( c1 -- t/f )  true if c1 is a valif name char > $20 < $7F
\   _forthpfa>nfa ( addr -- addr )
                              pfa>nfa for a forth word
\   _asmpfa>nfa ( addr -- addr )
                              pfa>nfa for an asm word
\   pfa>nfa ( addr -- addr )  gets the name field address (nfa) for a parameter field
                                  address (pfa)
\\  pfa? ( addr -- t/f)       true if addr is a pfa
```

Terminal

```
\\  crcl ( -- )               cr and clear the line (for an ansi terminal)
                               A simple terminal which interfaces to the a channel.
\   term ( n1 n2 -- )         n1 - the cog, n2 - the channel number
```

Miscellaneous

```
\\  lasm ( addr -- )          expects an address pointing to a structure in the
                                  following form empty long, long upper address of
                                  the assembler routine, long  lower address of the
                                  assembler routine - a series of longs which are the
                                  assembler codes

\\  !destination ( n1 n2 -- n1 )    set the d field of n1 with n2
\\  !instruction ( n1 n2 -- n1 )    set the i field of n1 with n2
\\  !source ( n1 n2 -- n1 )         set the s field of n1 with n2
```

These below are temporary variables, and by convention are only used within a word.
Caution, make sure you know what words you are calling.

```
: t0 98 parat ;
: t1 9A parat ;
: tbuf 9C parat ; \     0x20 (32) byte array overflows into numpad
```

Controlling one fast load at a time

```
wvariable fl_lock
wvariable fl_in

\   (flout)  ( -- )   attempt to output a character
\   (fl) ( --)        buffer input and emit
\   fl ( -- )         buffer the input and route to a free cog
\   fstart ( -- )     the start word ( This word is what the IP is set to on a reboot.)
```

More miscellaneous

```
\   fisnumber ( -- )   dummy routine s for indirection when float package is loaded

\   checkdict ( n -- ) make sure there are at least n bytes available in the dictionary
\   clabel ( cstr -- ) create an assembler constant at the current cog coghere



\   interpretpad ( -- ) interpret the contents of the pad
\   interpret ( -- )    the main interpreter loop
\   _wc1 ( x -- nfa )   skip blanks parse the next word and create a constant,
                         allocate a word, 2 bytes
\   >m ( n1 -- n2 )     produce a 1 bit mask n2 for position n
```

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

End of listing