Neon816

System Reference Manual

Neon816 System Reference Manual
Version 0.1

Table of Contents

NeonFORTH Language Reference

The Neon816 contains a full-featured Forth programming environment with a 16-bit cell length. Forth is a stack-based reverse polish language with support for high level programming, while maintaining an execution speed far faster than common token based languages.

Forth maintains two stacks, the Parameter and Return stacks. Unless otherwise specified, stack manipulations are specified in terms of the Parameter stack. The documentation of a Forth word will specify the input and output parameters, in this form:

```
DUP         ( a – a a )
```

This indicates that the DUP (duplicate) word takes one parameter on the stack, and replaces it with two copies of that same parameter.

Comments

There are two ways of defining comments in the Forth system.

```
\ Comment until end of line.
( This is a comment.  May span lines. )
```

Immediate Values

To push a numeric value on the stack, simply write the value:

```
5           ( Push the value 5 on the stack )
```

Constants

The following constants are defined by the Forth system for general use:

```
TRUE        ( – 65535 ; All bits are set, always true )
FALSE       ( – 0 ; All bits are clear, always false )
BL          ( – 32 ; Blank, the ASCII space character )
```

Stack Manipulation Words

This set of words manipulates the stack without any other side-effects.

```
DUP         ( a – a a )
DROP        ( a – )
```

```
SWAP      ( a b – b a )
OVER      ( a b – a b a )
PLUCK     ( a b c – a b c a )
ROT       ( a b c – b c a )
-ROT      ( a b c – c a b )
NIP       ( a b – b )
TUCK      ( a b – b a b )
```

Some of these also exist in double-word forms:

```
2DROP     ( a b – )
2DUP      ( a b – a b a b )
2SWAP     ( a b c d – c d a b )
```

There are special-purpose stack manipulation words as well:

```
PICK      ( … n2 n1 n0 q – … n2 n1 n0 nQ ; Duplicate from depth q )
?DUP      ( a – a ?a ; Duplicate a if a is not zero )
>R        ( a – ; Transfer a to the Return stack )
R>        ( – a ; Transfer a from the Return stack )
```

A set of special words are used to directly manipulate the stack pointer. These do not normally appear in user code, but are included for completeness.

```
0SP       ( – ; Clear the stack )
SP@       ( – addr ; Get parameter stack pointer )
SP!       ( addr – ; Set parameter stack pointer )
RP@       ( – addr ; Get return stack pointer )
RP!       ( addr – ; Set return stack pointer )
```

The default top of stack locations are stored in the following constants:

```
SP0       ( – addr ; Top of parameter stack )
RP0       ( – addr ; Top of return stack )
```

Sometimes it is helpful to measure the current stack depth, in cells:

```
DEPTH     ( – depth ; Compute the number of cells on the parameter stack )
```

Defining Words

STANDARD WORDS

The Forth system allows user-defined words. The most common way is this form:

```
: word ( Comments, normally describing the stack behavior of the word )
     \ User code here
;
```

It is also possible to create user-defined constant values or variable names. Constants push their value on the stack, while variables push the address of their data cell on the stack.

```
5 CONSTANT FIVE     ( Create a constant FIVE )
VARIABLE VAR        ( Create a variable VAR )
```

LOW-LEVEL WORDS

At the lowest level, the Forth program area is an array of program bytes that grows upward in memory.

The following variables maintain the state of memory:

```
VOCAB     ( VARIABLE ; First instruction of newest word )
TOP       ( VARIABLE ; Top of available memory )
```

The following words can be used to manipulate the program area:

```
HERE      ( – h ; Address of next program area byte )
FREE      ( – n ; Number of free bytes in program area )
ALLOT     ( n – addr ; Allocate next n bytes in program area )
,C        ( c – ; Push byte c to the program area )
,         ( w – ; Push word w to the program area )
,S        ( addr – ; Copy NUL-terminated string to program area )
```

New words may also be created using more advanced interfaces. The lowest level interface is HEADER, which creates a new word, and links it into the vocabulary, but does not add any contents. This can be useful for defining new words in native machine code. The CREATE word performs all of the same functions, and then adds instructions to push the address of the next byte of the program area.

```
HEADER name    ( Creates a new word name, with no content )
CREATE name    ( Creates a new word name, which pushes its body address )
```

USER-DEFINED DEFINING WORDS

This can be used to define user-defined defining words. This is most commonly used with the DOES> word, which defines the function of the body of the word created by the user-defined defining word. For example, VARIABLE could be defined as the following:

```
: VARIABLE CREATE 0 ,
    DOES> @ ;
```

CONTROLLING COMPILATION

The next group of words allow manipulating the most recently defined word. Words marked as immediate are executed when parsed in the instruction stream, even if a word is currently being compiled. They are most frequently used to implement control flow statements.

```
HIDE      ( Hides the most recent word from name lookup )
REVEAL    ( Allow the most recent word in name lookup )
IMMEDIATE ( Mark the most recent word as immediate )
```

It is possible to add the compiling semantics of a word, rather than the call to that word, into a word's definition. It is also possible to temporarily turn off compiling, for example to compute a value, and then later turn it back on and even insert the computed value into the compiled word.

```
POSTPONE word       ( compile word's compiling semantics )
]                   ( Turn off compilation )
[                   ( Turn on compilation )
LITERAL             ( n – ; Compile code to push n on the stack )
```

UNDEFINING WORDS

It is possible to delete all word definitions newer than any point in the dictionary, freeing their memory for further use:

```
FORGET word         ( – ; Forget word and all newer words )
```

Control Flow

CONDITIONAL STATEMENTS

The Forth system supports standard conditional expressions. Note that THEN terminates the statement block.

```
cond IF … THEN           ( Execute … if cond is true )
cond IF … ELSE … THEN    ( Execute first … if true, second … if false )
```

LOOPING STATEMENTS

There are two styles of looping in the Forth system. The first consists of the uncounted loops, and includes the following loop forms:

```
BEGIN … AGAIN                   ( Infinite loop )
BEGIN … cond UNTIL              ( Loop until cond )
BEGIN cond WHILE … REPEAT       ( Loop while cond )
```

The second style are the counted loops, and include the following loop forms:

```
m n DO … LOOP                   ( Loop from n to m )
m n ?DO … LOOP                  ( Loop from n to m, or zero times )
m n DO … p +LOOP                ( Loop from n to m, incrementing by p )
n TIMES word                    ( Execute word n times )
```

Counted loops provide easy access to the loop counters of the inner two loops:

```
I                               ( — n ; Loop count of inner loop )
J                               ( — n ; Loop count of outer loop )
```

EXIT STATEMENTS

To leave a word, use the EXIT word. If execution is currently inside a counted loop, use UNLOOP once per counted loop.

```
EXIT                ( Return from the current word )
UNLOOP              ( Clean up from a counted loop before EXIT )
```

EXECUTION STATEMENT

To directly execute a word or machine-language procedure, use the EXECUTE word.

```
EXECUTE             ( n — ; Execute address n )
```

Arithmetic

BASIC ARITHMETIC

The following words perform basic arithmetic operations:

```
1+          ( a — a+1 ; Increment a )
1-          ( a — a-1 ; Decrement a )
+           ( a b — a+b ; Add a and b )
-           ( a b — a-b ; Subtract b from a )
NEGATE      ( a — -a ; Negate a )
```

The following numbers provide 32-bit double cell and mixed width operations:

```
M+          ( al ah b — sl sh ; Add a and b )
```

```
D+        ( al ah bl bh – sl sh ; Add a and b )
D-        ( al ah bl bh – dl dh ; Subtract a and b )
DNEGATE   ( al ah – nl nh ; Double cell negate )
```

The following boolean operations are also available:

```
INVERT    ( a – ~a ; Invert a )
AND       ( a b – a&b ; Binary AND a and b )
OR        ( a b – a|b ; Binary OR a and b )
XOR       ( a b – a^b ; Binary XOR a and b )
```

Bit shifting is a useful operation, and is provided by these words:

```
2*        ( a – a*2 ; Shift a one place left )
2/        ( a – a/2 ; Arithmetic shift a one place right )
U>>       ( a n – a>>n ; Logical shift a n places right )
<<        ( a n – a<<n ; Shift a n places left )
8<<OR     ( a b – a|(b<<8) ; Combine bytes into word )
```

Words that compute the largest and smallest value are:

```
MIN       ( a b – min(a,b) ; Signed minimum of a and b )
MAX       ( a b – max(a,b) ; Signed maximum of a and b )
```

MULTIPLICATION/DIVISION

The Forth system provides both full width and truncating forms of both multiplication and division, for both signed and unsigned operations. The truncating forms are most commonly used as both input and output values are single cell:

```
*         ( a b – a*b ; Lower word of a*b for either signed or unsigned )
U/        ( n d – n/d ; Unsigned divide n by d )
/         ( n d – n/d ; Signed divide n by d )
```

The full width forms involve the use of double-cell numbers, for cases where 16 bit values are insufficient:

```
UM*       ( a b – ml mu ; 32-bit result of a*b unsigned )
M*        ( a b – ml mu ; 32-bit result of a*b signed )
UM/MOD    ( nl nh d – mod quo ; Unsigned divide 32-bit n by d )
FM/MOD    ( nl nh d – mod quo ; Signed divide 32-bit n by d )
```

COMPARISON

The following words perform comparison:

```
0=      ( a – a==0 ; Test if a is zero )
0<>     ( a – a<>0 ; Test if a is not zero )
0<      ( a – a<0 ; Test if a<0 )
=       ( a b – a=b ; Test if a is equal to b )
<>      ( a b – a<>b ; Test if a not equal b )
<       ( a b – a<b ; Test if a is less than b )
>       ( a b – a>b ; Test if a is greater than b )
<=      ( a b – a<=b ; Test if a is less than or equal to b )
>=      ( a b – a>=b ; Test if a is greater than or equal to b )
U<      ( a b – a<b ; Unsigned test if a is less than b )
U>      ( a b – a>b ; Unsigned test if a is greater than b )
```

Memory Access

To access memory or IO devices on Neon, a set of memory access words are provided. The normal 16-bit cell access functions can only reach the lower 64k of RAM, while the long forms can reach anywhere in memory or IO address space.

CELL ACCESS

```
C@      ( addr – val ; Read byte val from addr )
@       ( addr – val ; Read word val from addr )
C!      ( val addr – ; Write byte val to addr )
!       ( val addr – ; Write word val to addr )
+!      ( val addr – ; Add val to word at addr )
-!      ( val addr – ; Subtract val from word at addr )
LC@     ( addrl addrh – val ; Read byte val from long addr )
L@      ( addrl addrh – val ; Read word val from long addr )
LC!     ( val addrl addrh – ; Write byte val to long addr )
L!      ( val addrl addrh – ; Write word val to long addr )
```

COMPATIBILITY

For compatibility with other Forth systems, the following words are provided:

```
CELL    ( – 2 ; Unconditionally returns 2, bytes per cell )
CELLS   ( n – n*2 ; Multiplies n by 2, address from cells )
```

BLOCK OPERATIONS

The following words provide operations on memory blocks:

```
FILL    ( dest len char – ; Fill memory in bank 0 at dest with char )
MOVE    ( src dest len – ; Copy memory in bank 0 from src to dest )
```

String Operations

NUMERIC CONVERSION

The Forth system provides a suite of numeric conversion operations:

```
ISUNUM    ( c – cond ; Test if character c is an unsigned numeric digit )
ISNUM     ( c – cond ; Test if character c is a numeric digit or minus sign )
UATOI     ( addr – n ; Convert c-string at addr to unsigned integer )
ATOI      ( addr – n ; Convert c-string at addr to signed integer )
UITOA     ( n – addr ; Convert n to c-string in internal buffer )
```

Numerical conversion is controlled by the system radix setting, as provided by these words:

```
RADIX     ( VARIABLE ; The system number base, default 10 )
DECIMAL   ( - ; Set RADIX to 10, Immediate )
HEX       ( - ; Set RADIX to 16, Immediate )
```

Converting integers to strings uses an internal buffer. While no numeric conversions are in progress, this is also available as a small scratch buffer.

```
SCRATCH   ( – addr ; The address of the 64-byte numeric conversion buffer )
```

STRING COMPARISON

The Forth system also provides words for handling NUL-terminated (C-like) strings:

```
STRLEN    ( addr – len ; Compute the c-string length )
STRCMP    ( addr1 addr2 – eq ; Test c-strings for matching )
```

TEXT PARSING

A set of simple text parsing operations are provided by the Forth system. These generally operate on a NUL-terminated string, returning two NUL-terminated strings. This is done in-place, so the original string is modified by the operation.

```
WORD      ( addr – addr w-addr ; Parse a whitespace-delimited word )
SPLIT     ( addr c – addr w-addr ; Parse until delimiter character c )
```

Console IO

The Forth system provides standard words for console IO. Numeric conversion respects the current setting of RADIX. The following words write to the console:

```
EMIT      ( c – ; Write character c to the console )
CR        ( – ; Write CR/LF sequence to the console )
BS        ( – ; Erase the most recent character from the console )
SPACE     ( – ; Write a single space to the console )
TYPE      ( addr – ; Write a NUL-terminated string addr to the console )
.         ( n – ; Write the number n to the console )
U.        ( n – ; Write the unsigned number n to the console )
```

The following words read input from the console:

```
KEY?      ( – f ; Tests if input is waiting in the console buffer )
KEY       ( – k ; Wait and read key k from the console buffer )
ACCEPT    ( buf size – line ; Read line into buf with editing )
```

The following functions are provided for user convenience:

```
.S        ( – ; Print the complete contents of the parameter stack )
WORDS     ( – ; Print the names of every defined word )
```

Interpreter

The main Forth system interpreter is implemented in Forth. Many of these words are documented only for completeness, but some may be occasionally useful for other purposes.

The following represents the internal state of the interpreter:

```
STATE     ( VARIABLE ; True if compiling, False if interpreting )
TIB       ( – addr ; The address of the Terminal Input Buffer )
TIBPTR    ( VARIABLE ; The unparsed input line for the interpreter )
```

The interactive terminal stream can be parsed with the following words:

```
TIBWORD   ( – addr ; Parse a whitespace-delimited word from TIB )
TIBSPLIT  ( c – addr ; Parse until delimiter character c from TIB )
```

The interpreter dictionary lookup has the following interface:

```
FIND      ( name – addr ; Find the code body of a word from c-string name )
‘ name    ( – addr ; Find the code body of name )
>NAME     ( addr – name ; Offset to word name from code body addr )
```

The interpreter can be manually invoked. Note that while the interpreter will normally return to the caller upon normal completion, it will instead call ABORT on errors. These are the main interpreter entry points:

```
INTERPRET ( sbuf – ; Interpret c-string sbuf )
QUIT      ( – does not return ; Return to interactive interpreter )
ABORT     ( – does not return ; Clear the stack and call QUIT )
COLD      ( – does not return ; Initial entry point of the Forth system )
```

Hardware Access

MIDI INTERFACE

The following words allow access to the hardware MIDI ports:

```
MIDI!     ( c – ; Send byte c to the MIDI out port )
MIDI?     ( – f ; Test if input is waiting in the MIDI input buffer )
MIDI@     ( – c ; Wait and read byte c from the MIDI input buffer )
```

The MIDI protocol itself is outside the scope of this document. Refer to the MIDI Association's documentation for more information.

PS/2 INTERFACES

The following words allow low-level access to the hardware PS/2 ports. The names indicate the intended use of the ports (keyboard vs mouse), but the two ports are electrically and logically identical. It is valid to connect a keyboard to the mouse port or a mouse to the keyboard port if using the low-level interface.

```
PS2K!     ( c – ; Send byte c to PS/2 port 1 )
PS2K?     ( – f ; Test if input is waiting in the input buffer of PS/2 port 1 )
PS2K@     ( – c ; Wait and read byte c from the input buffer of PS/2 port 1)
PS2M!     ( c – ; Send byte c to PS/2 port 2 )
PS2M?     ( – f ; Test if input is waiting in the input buffer of PS/2 port 2 )
PS2M@     ( – c ; Wait and read byte c from the input buffer of PS/2 port 2)
```

REAL-TIME CLOCK

The following words allow access to the real-time clock hardware.

```
SETRTC    ( day hour min sec ms us – ; Set the RTC )
GETRTC    ( – day hour min sec ms us – ; Read the RTC )
```

I²C INTERFACES

The following words allow access to the I²C controller hardware:

```
I2C2START ( – ; Send a START condition on I²C controller 2 )
I2C2STOP  ( – ; Send a STOP condition on I²C controller 2 )
```

```
I2C2!     ( byte – ; Send a byte on I²C controller 2 )
I2C2@+    ( – byte ; Receive a byte on I²C controller 2 with acknowledge )
I2C2@     ( – byte ; Receive a byte on I²C controller 2 without acknowledge )
```

Since I²C controller 2 is connected to both the DVI encoder and the DVI port's DDC lines, it is possible to use it to read the EDID ROM present in a DVI or HDMI monitor:

```
DUMPEDID  ( – ; Dump the first 256 bytes of the EDID ROM )
```

SPI INTERFACES

The following words allow access to the Serial Peripheral Interface (SPI) controllers:

```
SPI2INIT  ( – ; Set SPI controller 2 to default configuration )
SPI2START ( – ; Enable CS for SPI controller 2 )
SPI2STOP  ( – ; Disable CS for SPI controller 2 )
SPI2!     ( byte – ; Send a byte on SPI controller 2 )
SPI2@     ( – byte ; Receive a byte while shifting out 0 on SPI controller 2 )
```

JOYSTICK INTERFACES

The Forth system provides easy access to joysticks of either Atari standard or Sega standard. The joystick controller is connected to SPI controller 2, but this is entirely abstracted by the higher-level joystick interface.

```
JOYSETUP  ( – ; Configure both joystick ports with user input )
JOY@      ( joyno – joyval ; Read joystick joyno )
JOYTEST   ( – ; Interactive joystick testing routine.  Press a key to end )
```

The joystick driver automatically converts all joysticks into a single 16-bit button field map:

Byte	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Low Byte	0	0	0	0	0	0	0	0
High Byte	0	0	0	0	0	0	0	0

Alphabetical Index