Logarithmic Number Representation in Forth

Dennis L. Feucht
Innovatia Laboratories

Abstract

The log-point representation is an alternative to floating point. Whereas floating point combines linear and logarithmic scales, log point is purely logarithmic. It maintains a constant precision over its range, thus minimizing the accumulation of round-off error. Log-point characteristics are demonstrated, compared to floating-point, and a Forth implementation of a log-point word-set is provided.

Introduction

The Forth language provides fixed-point numerical capabilities, but many scientific and engineering problems require an extended-range representation. Floating-point representation has been the most popular, and Forth floating-point word-sets have been developed. An alternative representation, log point, is described here, and an implementation of it in Forth is given. Fixed and floating-point representations will first be briefly reviewed, leading to the log-point concept.

Arithmetic operations in Forth use a fixed-point two's-complement representation of numbers. Not every number can be represented by fixed point-only integers within a 16-bit range. Rational numbers can be easily accommodated by assuming the binary-point is at a given bit position. This results in integer and fractional fields within the 16-bit number, and can be expressed as $Q = I \cdot 2^{-i}$, where I is an integer represented in fixed point with a binary point at the ith bit, resulting in rational number Q. The fixed-point arithmetic words of Forth can be used with rational numbers with some modification. Products must be scaled by multiplying them by 2^i and quotients by 2^{-i}. Forth provides the word `*/` to do this without loss of precision.

For calculations with a dynamic range greater than 16 bits, some other representation is needed. Double numbers can extend the range of fixed point to 32 bits, but this is often too limiting for many scientific and engineering applications. The conventional solution to the limited-magnitude problem has been to use a floating-point representation.

Floating vs. Fixed-Point Representation

Floating point (or scientific notation) greatly extends the range of numbers it represents. Instead of integer and fractional fields, floating point splits the n-bit cell into mantissa and exponent fields instead. This scheme is a combination of linear (mantissa) and logarithmic (exponent) representations. The exponent provides increased range at the expense of fewer bits (and thus less precision) in the mantissa. However, over the domain of represented numbers, floating point usually has better precision than fixed point because range error is limited to an order of magnitude; that is, the mantissa has (for decimal) a 10:1 range over the entire floating-point range. Because of this, small numbers can maintain their significant figures (their precision) by scaling them with the exponent. Fixed-point numbers suffer from range error increasingly as the number becomes smaller. For example, the fixed-point number 1000, representing 1.000, has three significant figures while 1 has almost none. Range error increases as

the reciprocal of the number so that nx is n times as precise as x. To achieve precision comparable to floating point, many more bits in the represented number are required.

In arithmetic operations, fixed point is exact for addition (or subtraction) but rounds or truncates the least significant figure (LSF) when multiplying (or dividing). Floating point rounds for both operations. Fixed point, because of its additive exactness, is optimal for counting, which historically must have been the first use for numbers. [SIM80] In computing, it is preferred for counting applications such as indexing in an iterative loop. It appears then that both fixed point and an extended-range representation are desirable for scientific and engineering numeric computing.

Log-Point Characteristics

Log point is an extended-range representation which can be described starting with floating point. Only the sign of the mantissa is kept; the magnitude is included in the exponent, which is made a rational number. The original number is represented by its logarithm to a given base. Since the logarithm is defined only for positive numbers, the sign of the mantissa is retained to provide signed representation. The resulting log-point number has two signs: of the number itself and of the exponent. A log-point number can be expressed as: $\pm b^{\pm q}$, where b is the positive integer base of the log-point representation and q is the rational number signed exponent, with whole and fractional parts.

Interestingly, log point has no zero since the logarithm is undefined there. It can be approached to within the precision of the representation but cannot be represented purely as a log-point number. (In the implementation given here, zero is quite easily and naturally represented as a simple extension to pure log point.) Clearly, log point is not optimal for counting, but it is for ratioing, which occurs frequently in scientific and engineering computations. Multiplication is exact since the product of two numbers is the fixed-point sum of their exponents. Unlike fixed point, addition rounds and is the hardest to perform. However, tabular addition is quite feasible, requiring a one-dimensional look-up table. Tabular fixed-point multiplication requires a two-dimensional table with a correspondingly larger memory requirement.

Unlike both fixed and floating point, log point has a constant precision over its entire range. Because the log-point scale is logarithmic, the step size between least significant states is not regular as with fixed point but increases with the magnitude of the number. Since this increase is exponential, the step size remains a constant fraction of the number. Consequently, the precision of the number is the same as all other numbers over the log-point range.

The particular log point implementation chosen here for 16-bit arithmetic is:

$$ms\ b^{\{es\ w.f\}}$$

where:

b = base of the representation

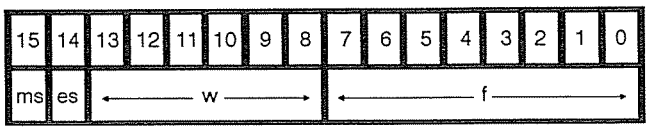
ms = sign of the mantissa (that is, of the number itself) in two's-complement form (ms = 1 for negative numbers)

es = sign of the exponent in offset binary form (es = 0 for negative numbers.)

w = whole part of the exponent, with length of w bits

f = fractional part of the exponent, with length of f bits

- For this implementation, $b = 2, f = 8$, and $w = 6$:



Log Point Range and Precision

The maximum representable magnitude occurs with all log-point bits set to one, or:

$$\begin{aligned}\text{maximum magnitude} &= 2^{(2^w-1)+(2^f-1)/2^f} \\ &= 2^{(2^w-2^{-f})} \cong 2^{2^w}\end{aligned}$$

The minimum magnitude is:

$$\text{minimum magnitude} = 2^{-(2^w-2^{-f})} \cong 2^{-2^w}$$

The range of the magnitudes is the ratio:

$$\begin{aligned}\text{range} &= \text{max magnitude} / \text{min magnitude} = 2^{2^{w+1}-2^{-(f-1)}} \\ &\cong 2^{2^{w+1}} \cong 10^{2^{w+1}/3.32}\end{aligned}$$

For $w = 6$, the range is about 3.4×10^{38} or ± 19 decimal orders of magnitude. For 32-bit log point with $w = 10$ and $f = 20$, the range is $10^{\pm 308}$.

- The exponent is expressed in offset binary, which is two's-complement offset by half the number range. The smallest number is binary 00...00 and the largest is 11...11. For m bits, the smallest is $-2^{(m-1)}$ and the largest is $2^{(m-1)} - 1$. To convert from two's complement to offset binary, add binary 10...00 to the two's-complement number. This has the effect of merely complementing the sign bit (the MSB). Zero in two's complement is 00...00 and in offset binary is 10...00.

The precision of log point can be found by first considering calculation of fractional step sizes. Since the scale is non-linear, an increment of a least significant bit (LSB) is not exactly the same size as a decrement of an LSB from the same state. Since the scale is logarithmic, the incremental step will be larger than the decremental step. As $f \to \infty$, the incremental and decremental step sizes converge to each other. They are:

$$\begin{aligned}\text{incremental step size} &= 2^{2^{-f}} - 2^0 = 2^{2^{-f}} - 1 \\ \text{decremental step size} &= 2^0 - 2^{-2^{-f}} = 1 - 2^{-2^{-f}}\end{aligned}$$

The incremental step size will always be the maximum step size. With round off, the maximum log-point error will be:

$$\text{maximum error} = \text{maximum step size} / 2$$

Let X, a log-point number, represent x, where $x = 2^X$. The average step size can be taken as the ratio:

$$\text{avg fractional step size} = dx/dX = 2 \cdot \ln 2 = \ln 2 \cdot 2^{-f}$$

For $f = 8$, the average step size is 0.00271.

With uniformly distributed error (on a log scale):

$$\begin{aligned}\text{error(rms)} &= \text{max step size} / \sqrt{12} \\ &\cong (2^{2^{-f}} - 1)/3.46\end{aligned}$$

For $f = 8$, error $\cong \pm 0.04\%$.

Comparison of Bit Efficiency With Floating Point

A number representation makes more efficient use of bits if it provides more range and precision concurrently. Log point is somewhat more efficient than floating point. [LEE77][EDG79] For FFT transforms of 256 to 1024-point sequences, for a given numeric word length, the average error of log point is about an order of magnitude better than floating point. [SWA83] For 32-bit word lengths, log point has a range comparable to floating point, with 8– rather than 7+ digits of precision.

Sixteen Bit Log Point Implementation in Forth

The implementation of log-point arithmetic given here is written in Forth-83. The word set provides the following capability:

1. Input a number string and convert it to a log-point number.
2. Output a log-point number as a formatted ASCII string.
3. Perform common arithmetic operations.
4. Generate common transcendental functions.

The Mathematical Basis for Log-Point Arithmetic

To develop the mathematics of the log-point algorithms, numbers will be denoted by lower-case letters, such as x and y, and their log-point representations will be denoted by upper-case letters, X and Y. Thus, $x = 2^X$ and $X = \lg x$, where lg is $\log_2$.

To multiply two numbers x and y is to add their log-point representations, X and Y:

$$xy = 2^X \cdot 2^Y = 2^{X+Y}$$

Or, to take the square root of x:

$$x^{1/2} \rightarrow \lg x^{1/2} = \frac{1}{2} \cdot \lg x = \frac{X}{2}$$

where X is a rational fixed-point number. Fixed-point operations for $\lg x$, 2^x, and 2^{-x} are needed to implement log point from the fixed-point operations of Forth. As a side benefit, these functions can be used in fixed-point arithmetic.

Converting Number Strings to Log-Point Numbers

Number strings are of the form:

$$Sw.f\hat{}se$$

where S is optionally a minus sign, w a string of digits in the current base representing the whole part of the number, f the fraction, s the optional exponent minus sign, and e the exponent followed by a Forth delimiter. The Forth word which reads a number string from the input stream is **REAL**. It calls **LVAL** to do the actual conversion. **LVAL** converts Sw, f, and se in turn, parsing the point and exponent sign (^) as it goes. These signs are optional when f and se are, respectively, omitted. The only requirement is that either w or f be given.

To convert the string while parsing it, **LVAL** uses four numbers which become individually available: w, the whole number, f, the fraction expressed as a whole number, n, the number of digits in the fraction, and b, the current base, found in the Forth variable **BASE**. Defining the log-point addition function as:

$$X \; \mathbf{L+} \; Y = \lg(x + y)$$

Input conversion becomes:

$$w + fb^{-n} \rightarrow \lg(w + fb^{-n}) = W \text{ L+ } \lg fb^{-n} = W \text{ L+ } (F + (-n)\lg b)$$

The last term is carried out using fixed-point multiplication with scaling and the fixed-point word LOG2, which takes the log of a whole number by first scaling it using shifting operations before passing it to LOG2*. LOG2* has a domain of [1,2) and a range of [0,1).

Log Point Addition

Just as multiplication is difficult in fixed-point arithmetic, so addition is in log point. Four-quadrant addition is performed by L+ which uses the pair of functions SL+ and SL- to add directly using X and Y without having to convert either of them to x or y. SL+ and SL- are defined as:

$$\text{SL+}(z) = \lg(1 + 2^{-z})$$
$$\text{SL-}(z) = \lg(1 - 2^{-z})$$

Then,

$$x + y = 2^X + 2^Y = 2^X(1 + 2^{Y-X}) \rightarrow \lg 2^X(1 + 2^{Y-X})$$
$$= X + \text{SL+}(X - Y) = X \text{ L+ } Y$$

For subtraction,

$$x - y \rightarrow X + \lg(1 - 2^{Y-X}) = X + \text{SL-}(X - Y)$$

The domain of SL+ is $(-\infty, +\infty)$ while SL- is defined for positive numbers only. Also, SL+ is asymptotic to $y = -x$ as $x \rightarrow -\infty$. In the Forth word L+, decisions are made that ensure that the arguments of SL+ and SL- are always positive. Then SL+ need only be computed over the interval $[0, \infty)$. For tabular implementation of SL+, this halves the table size.

The eight branches in L+ depend on the signs of X, Y, and $(X - Y)$. SL+ is called when the magnitudes of X and Y are added, and SL- when subtracted. In L+, SL+ and SL- are always added to the larger of X and Y. When subtracting, since SL- is negative, it reduces the larger number, and when adding, the positive SL+ increases the larger number. L- negates the second argument and invokes L+.

Because SL$\pm(z)$ both approach zero as $z \rightarrow \infty$, tabulating for large z results in long sequences of the same numerical value. About 75% of each table consists of these slowly changing sequences. Table size could be reduced by about 67% by run-length coding the asymptotic tails of SL+ and SL-.

SL$\pm$ are calculated in the given implementation using fixed-point arithmetic and based on the following construction. Let the log-point argument to SL+ consist of whole and fractional parts w and f, respectively. Then,

$$\lg(1 + 2^{-(w+f)}) = \lg 2^{-w}(2^w + 2^{-f}) = -w + \lg(2^w + 2^{-f})$$

In the Forth words for SL+, since the fraction, f, is the least significant byte of the 16-bit number cell, w and f are conveniently separated by the word SPLIT. (See glossary for definition of SPLIT.) D+ adds 2^w and 2^{-f}. Now let $(2^w + 2^{-f})$ be represented as $m2^e$, where $1 \leq m < 2$. Then,

$$-w + \lg(2^w + 2^{-f}) = -w + \lg(m2^e) = -w + \lg m + e$$

Since m is within the domain of LOG2*, finishing the calculation is straightforward though scaling is involved.

- A simple method for rounding a number being divided by n is to divide by $n/2$, increment the result, and divide by two again. Doing this is the equivalent of adding 1/2 to the result, thus rounding instead of truncating.

Numeric Output Conversion

Converting log-point numbers to ASCII character strings is performed by the word **LSTR$**. It calls **L>E** which does the bulk of the conversion, while the rest of **LSTR$** creates the formatted string. Then **L.** prints the resulting string. **LSTR$** computes the number of significant figures possible and prints all figures with any significance. It uses **#DIGITS** to determine the number of digits that should be in the fraction of the mantissa.

L>E converts a log-point number to a mantissa and exponent in the current base. The mantissa ranges from one to the base minus a LSF. Converting a number to a mantissa and exponent is based on the following construction which uses four numbers, just as **LVAL** did. The only difference is that W and F are the whole and fractional parts of the log-point number. The current base (> 0) is b, and n fixes the location of the binary point in the number itself. (For this implementation, $n = 8$.) The construction is:

$$W + F2^{-n} \rightarrow 2^{(W + F2^{-n})} = b^{(W + F2^{-n})/\lg b}$$
$$= b^{W}/\lg b \cdot b^{F2^{-n}}/\lg b$$
$$\text{exponent} = W/\lg b, \text{ mantissa} = b^{F2^{-n}}/\lg b$$

The exponent in the expression for the mantissa is then broken into whole and fractional parts:

w = whole part of fraction
f = fractional part of fraction
$\text{mantissa} = 2^{(F2^{-n}/\lg b)} = 2^{w+f} = 2^{w}2^{f}$

Implementing Zero and Relational Operators

Throughout the log-point word-set, zero is handled as a special case. The log-point number 00...00 intentionally has the smallest magnitude possible by deliberate use of offset binary. This conveniently places the state of the number which is closest to zero at the same state as the fixed-point zero. Thus, zero, as a special case in log point, can be easily detected using Forth fixed point word **0=** .

Another advantage to offset binary is that the relational operators >, =, < can be the Forth fixed-point operators. The sign of the mantissa in the MSB is two's complement, and the magnitude of the number is ordered the same as an unsigned fixed-point number would be since the exponent sign can be regarded as the MSB.

A Limitation of Log Point

Tabular implementation of **SL±** gives log point its speed appeal. Since the four basic arithmetic functions execute on the order of fixed-point addition, log point with tabular addition can exceed fixed-point speed. For 16-bit implementations, the table size is feasible. As the log-point word length is extended, the table size becomes unwieldy. For long word lengths, **SL+** must be computed and speed suffers, but the other advantages remain.

References

[EDG79] Albert D. Edgar, Samuel C. Lee, "FOCUS Microcomputer Number system", *Comm. ACM*, vol. 22, no. 3, MAR 1979, p. 166ff.

[LEE77] Samuel C. Lee, Albert D. Edgar. "The FOCUS Number System", *IEEE Trans. on Computers*, vol. C-26, no. 11, NOV 1977, p. 1167ff.

[SIM80] Kenneth A. Simons, "N-Logs: A New Number Language for Scientific Computers", *Dr. Dobbs Journal*, no. 50, NOV/DEC 1980, p. 4ff.

[SWA83] Earl E. Swartzlander, et. al., "Sign/Logarithm Arithmetic for FFT Implementation", *IEEE Trans. on Computers*, vol. C-32, no. 6, JUN 1983, p. 526ff.

- References on fixed-point log and exp algorithms:

Nathaniel Grossman, "Fixed Point Logarithms", *Forth Dimensions*, vol. 5, no. 5, JAN/FEB 1984, p. 11ff

R.J. Linhardt, H.S. Miller, "Digit-by-Digit Transcendental Function Computation", *RCA Review*, JUN 1969, p. 209ff.

Log-Point Forth Word Glossary

This glossary gives the stack activity and a brief description of the function of the words in the Forth log-point word-set. The abbreviation ^ @ indicates the location of the fractional point (that is, decimal point for base ten) in a rational number.

#DIGITS (— n)

Calculates n, the number of significant figures possible for a given base; used by **L>E** and **LSTR$**.

-ADJUST (n1 — n2)

Adjusts from binary to decimal point using the constant **1ØK** to set the decimal point for four fractional digits. n1 ^ @ **ONE**.

-EXP2* (n1 — n2)

Performs 2^{-n1}, where n1 ^ @ **ONE** and $0 < n1 \leq 1$. n2 ^ @ **ONE** and $1/2 \leq n2 < 1$.

1ØK (— n)

A constant with value 10^4 used to adjust decimal point in **ADJUST** and **-ADJUST**.

L2/ (n1 — n2)

Performs a logical right shift on n1.

ADJUST (n1 — n2)

Adjusts from decimal to binary point, accepting a number for which the four rightmost digits are fractional, and adjusts it for a binary point at **ONE**.

D2/ (dn1 — dn2)

Performs a logical right shift on double number dn1.

D2* (dn1 — dn2)

Performs a double number doubling of dn1.

DNORM< (dn1 — dn2)

Performs a double number normalization of dn1 by shifting dn1 left until dn2 has a binary 1 whole number with ^ @ 2^{26} (^ @ **ONE** in the most significant byte of dn2).

EFACTORS (n1 — n2)

A table that takes index n1 and returns a value, n2; used by **EXP2***.

EXP2* (n1 — n2)

2^{n1} function that takes n1 ^ @ **ONE**, where $0 \leq n1 < 1$ and returns n2 ^ @ **ONE** with $1 \leq n2 < 2$. **EXP2*** is a fixed-point algorithm described in the references.

FIXED (ln — dn)

Converts a log-point number to a fixed-point whole double number. Any fractional parts of ln are truncated.

FLOAT (dn — ln)

Converts a whole double number to a log-point number.

L^ (ln1 ln2 — ln3)

Raises ln1 to the ln2 power and returns a log-point number, ln3.

L* (ln1 ln2 — ln3)

Multiplies two log-point numbers.

L+ (ln1 ln2 — ln3)

Adds two log-point numbers.

L- (ln1 ln2 — ln3)

Subtracts log-point number ln2 from ln1.

L. (ln —)

Prints the log-point number, ln, using **LSTR$**. The number is formatted in scientific notation and is followed by a space.

L/ (ln — e m)

Converts a log-point number, ln, to an exponent, e, and a mantissa, m. The exponent is an integer and the mantissa is a rational number with ^ @ $2^{\text{the value returned by \#DIGITS}}$

LABS (ln1 — ln2)

Returns absolute value of ln1.

LFACTORS (n1 — n2)

Similar to **EFACTORS** - a table that takes index n1 and returns a value used in **LOG2*** and **-EXP2***.

LNEGATE (ln1 — ln2)

Negates a log-point number.

LOG# (n1 — n2)

Marks the beginning of the log-point word-set in the Forth dictionary and, when invoked, prints version information.

LOG2 (n1 — n2)

Calculates the lg ($\log_2$) of n1 in fixed-point arithmetic. It extends the range of **LOG2***. n1 is a whole number and n2 ^ @ 2^8 (the least significant byte is the fraction, as in log-point representation).

LOG2* (n1 — n2)

Calculates the lg of a fixed-point number, n1, where n1 ^ @ **ONE** and $1 \leq n1 < 2$. n2 ^ @ **ONE** and $0 \leq n2 < 1$. The algorithm is described in the references.

LSQR (ln1 — ln2)

Takes the square of ln1.

LSQRT (ln1 — ln2)

Takes the square root of ln1.

LSTR$ (ln — str)

Takes a log-point number and converts it to a string, str, where str → a u; a = address of the first character of string, u = number of characters in string. The number of fractional digits is determined by **#DIGITS**. A period (.) and a caret (^) denote fractional point and exponent, respectively.

LVAL (str — ln)

Takes a string [see **LSTR$**] and converts it to log point. A minus sign can optionally preceed the whole number or the exponent. Either the whole number or fraction must be given. The exponent and whole number or fraction are optional.

MLOG2 (dn1 — n2)

A mixed-number word that takes the logarithm of double number dn1, which is a whole number. n2 ^ @ 2^8 as for log point.

N. (n —)

Prints an adjusted rational number with ^ @ **ONE** to four fractional decimal digits. Useful for displaying results of **LOG2***, **EXP2***, or **-EXP2***.

NORM< (n1 — e n2)

Used by **LOG2** to normalize a whole number to be within the domain of **LOG2***: $1 \leq n1 < 2$. e is a whole number binary exponent. n2 ^ @ **ONE**.

ONE (— n)

Constant that fixes the binary point for **LOG2***, **EXP2***, and **-EXP2*** at 2^{13}. This allows maximum precision without overflow for these functions.

REAL (— ln)

Converts the next word in the input stream to log point using **LVAL**. When compiling, it compiles a literal log-point number.

SL+ (n1 — n2)

A function used by **L+** to add. It is $\lg(1+2^{-n1})$, where n1 and n2 ^ @ 2^8, and n2 ≥ 0.

SL- (n1 — n2)

Similar to **SL+**; used by **L+** to subtract. It is $\lg(1-2^{-n1})$, where n1 and n2 ^ @ 2^8, and n1, n2 ≥ 0.

^2 (n1 — n2)

A table used by **SL+** and **SL-** to return a whole number power of two. n2 = 2^{n1}.

Non-Standard Words

The given implementation is written in Forth-83, but uses the following non-83-standard words.

2* (n1 — n2)

Arithmetic left shift.

2/ (n1 — n2)

Arithmetic right shift.

<SHIFT (n1 n2 — n3)

Performs an arithmetic left shift of n2 bits on n1.

```
: <SHIFT  0 ?DO 2* LOOP ;
```

SHIFT (n1 n2 — n3)

Performs an arithmetic right shift of n2 bits on n1.

```
: >SHIFT  0 ?DO 2/ LOOP ;
```

?DO (—)

A variation on **DO** that immediately leaves the loop if the top two arguments on the stack are equal. An alternative construct for a zero initial index value would be:

```
?DUP IF Ø DO ... LOOP THEN
```

COMBINE (n1 n2 — n3)

Joins the low-order bytes of n1 and n2 into a single cell, n3, where the low-order byte of n3 is from n1.

```
: COMBINE 8 <SHIFT SWAP 255 AND OR ;
```

NIP (n1 n2 — n3)

Removes second item from stack.

```
: NIP SWAP DROP ;
```

SPLIT (n1 — n2 n3)

The inverse operation of **COMBINE** ; n1 is split into two bytes. The low-order byte becomes the low-order byte of n2 and the high-order byte becomes the low-order byte of n3. The high-order bytes of n2 and n3 are zero.

```
: SPLIT DUP 255 AND SWAP 8 >SHIFT 255 AND ;
```

TABLE (n1 — n2)

A defining word that creates a header at compile time and at run time uses n1 as an index from its body to an address from which it fetches a number and places it on the stack.

```
: TABLE CREATE DOES> SWAP 2* + @ ;
```

```
SCR# 39
\ Log-Point Number Words - Load Screen

DECIMAL

ONLY FORTH ALSO DEFINITIONS

: LOG-PT ." 19 MAY 1985" ;

1 17 +THRU

UNNEST

: >SHIFT Ø ?DO 2/ LOOP ;
: <SHIFT Ø ?DO 2* LOOP ;


SCR# 4Ø
\ Log-Point Number Words

8192  CONSTANT ONE

 ( dn1 -> dn2 )
: D2/ DUP 1 AND >R 2/ SWAP 2/ R>
     IF 32768 OR ELSE 32767 AND THEN SWAP
;
 ( dn1 -> e n2 ) ( dn1 & n2 ^ @ ONE; dn1 >= Ø )
 : MNORM> 2DUP OR
     IF Ø >R
          BEGIN 2DUP [ ONE 2* ] LITERAL Ø D< Ø=
          WHILE D2/ R> 1+ >R
          REPEAT DROP R> SWAP
     THEN
;


SCR# 42
\ Log-Point Number Words

TABLE LFACTORS  ( 13 factors beginning with i = 2 )

3400 , 1578 , 763 , 375 , 186 , 93 ,
46 , 23 , 12 , 6 , 3 , 1 , 1 ,

 ( n1 -> n2 ) ( binary point is at ONE )
           ( Ø <= n1 < 1 ; 1/2 <= n2 < 1 )
: -EXP2* ONE 2
     BEGIN DUP 2- LFACTORS DUP >R 3 PICK SWAP <
          IF R> DROP 1+
          ELSE ROT R> - -ROT 2DUP >SHIFT ROT SWAP - SWAP
          THEN DUP 13 =
     UNTIL DROP NIP
;
```

```
SCR# 42
\ Log-Point Number Words


 ( n1 -> n2 )  ( Takes ADJUSTed n1; returns ADJUSTed n2 )
: LOG2* Ø 2 ROT
     BEGIN DUP DUP 3 PICK >SHIFT - DUP ONE <
          IF DROP SWAP 1+ SWAP
          ELSE NIP ROT 2 PICK 2- LFACTORS + -ROT
          THEN OVER 13 =
     UNTIL 2DROP
;



SCR# 43
\ Log-Point Number Words

 ( dn1 -> e dn2 )
: DNORM< 2DUP OR
     IF Ø >R
          BEGIN DUP ONE <
          WHILE 2* OVER 32768 AND
               IF 1 OR THEN SWAP 2* SWAP R> 1+ >R
          REPEAT R>
     ELSE Ø
     THEN -ROT
;


SCR# 44
\ Log-Point Number Words

 ( Used by L+ )

TABLE ^2 1 , 2 , 4 , 8 , 16 , 32 , 64 , 128 , 256 ,
     512 , 1Ø24 , 2Ø48 , 4Ø96 , 8192 , 16384 , 32768 ,

 ( n1 -> n2 ) ( n1, n2 >= Ø; ^@ 2^8 )
: SL+ SPLIT >R 32 * -EXP2* Ø R@ ^2 ONE *D D+
     MNORM> LOG2* 16 / 1+ 2/ SWAP R> - 256 * +
;

: SL- SPLIT >R 32 * -EXP2* Ø DNEGATE R@ ^2 ONE *D D+
     DNORM< NIP LOG2* 16 / 1+ 2/ SWAP 16 - R> + -256 * +
;
```

```
SCR# 45
\ Log-Point Number Words
HEX

: LABS 7FFF AND ;
: LNEGATE DUP IF 8000 XOR THEN ;

: L* 2DUP AND IF + 4000 - ELSE AND THEN ;
: L/ OVER IF - 4000 + ELSE DROP THEN ;

CODE L2/ BOT 1+ LSR  BOT ROR  NEXT JMP  C;
: LSQRT DUP IF 4000 + L2/ THEN ;

\ : LSQRT DUP IF 4000 + 2/ 7FFF AND THEN ;

: LSQR DUP IF 2* 4000 - THEN ;
DECIMAL

SCR# 46
\ Log-Point Number Words

 ( X Y -> Z )
: L+ 2DUP 0= SWAP 0= OR 0=
     IF 2DUP - >R OVER 0<
          IF DUP 0<
               IF R> DUP 0< ( X<0; Y<0 )
                    IF NEGATE SL+ + NIP
                    ELSE SL+ NIP +
                    THEN
               ELSE R> DUP 0< ( X<0; Y >= 0 )
                    IF 32768 + SL- NIP + ( 32768 = 8000 HEX )
                    ELSE 32768 SWAP - SL- + NIP
                    THEN
               THEN

SCR# 47
\ Log-Point Number Words

          ELSE DUP 0<
               IF R> DUP 0< ( X>=0; Y<0 )
                    IF 32768 + SL- NIP +
                    ELSE 32768 SWAP - SL- + NIP
                    THEN
               ELSE R> DUP 0< ( X>=0; Y>=0 )
                    IF NEGATE SL+ + NIP
                    ELSE SL+ NIP +
                    THEN
               THEN
          THEN
     ELSE OR
     THEN
;
```

```
SCR# 48
\ Log-Point Number Words

: L- LNEGATE L+ ;

 \ = < > Ø= Ø< Ø> are same as fixed-point

1ØØØØ CONSTANT 1ØK

 ( n1 -> n2 )
: ADJUST ONE 1ØK */ ; \ Adjust from decimal to binary point.

 ( n1 -> n2 )
: -ADJUST 1ØK ONE */ ;

 ( n -> )
: N. -ADJUST S>D <# # # # # ASCII . HOLD #S #> TYPE ;

SCR# 49
\ Log-Point Number Words

 ( n1 -> e n2 )
: NORM< DUP
     IF Ø SWAP
          BEGIN DUP ONE <
          WHILE SWAP 1+ SWAP 2*
          REPEAT
     ELSE Ø
     THEN
;

 ( n1 -> n2 ) ( n2 has binary point at 2^8 - LSByte is fraction)
              ( 32 / shifts bp of log2* set by ONE )
              ( n1 is a positive integer )
              : LOG2 NORM< LOG2* 16 / 1+ 2/ 13 ROT - COMBINE ;

SCR# 5Ø
\ Log-Point Number Words

 ( dn1 -> n2 ) ( dn1 is a positive integer; n2 ^ 2^8 )
: MLOG2 DNORM< NIP LOG2* 16 / 1+ 2/ 29 ROT - COMBINE ;

 ( dw -> 1# ) ( dw is a whole number: dw >= Ø )
: FLOAT 2DUP OR IF DUP >R DABS MLOG2 16384 XOR R> Ø<
     IF LNEGATE THEN ELSE DROP THEN
;

 ( str -> 1# )
: LVAL >R PAD 1+ R@ CMOVE BL PAD 1+ R> + C! ( move str to PAD )
     Ø Ø PAD DUP 1+ C@ ASCII - = DUP >R ( save sign )
     IF 1+ THEN CONVERT >R DABS FLOAT R> DUP C@ ASCII . =
     IF DUP 1+ >R Ø Ø ROT CONVERT >R FLOAT
          R> R> OVER >R SWAP - 2DUP AND
```

```
SCR# 51
\ Log-Point Number Words

          IF BASE @ LOG2 * + LABS L+ ELSE 2DROP THEN R>
     THEN OVER ( is mantissa non-zero? )
     IF DUP C@ ASCII ^ =
          IF Ø Ø ROT DUP 1+ C@ ASCII - = DUP >R IF 1+ THEN
               CONVERT -ROT DROP R> IF NEGATE THEN SWAP >R
               BASE @ LOG2 * + R>
          THEN C@ BL OR BL - ABORT" ?" R> IF LNEGATE THEN
     ELSE DROP R> DROP
     THEN
;

: REAL BL WORD COUNT LVAL STATE @
     IF [COMPILE] LITERAL THEN
; IMMEDIATE

SCR# 52
\ Log-Point Number Words

TABLE EFACTORS ( 13 FACTORS)

2637 , 1392 , 716 , 364 , 183 , 92 ,
46 , 23 , 12 , 6 , 3 , 1 , 1 ,

 ( n1 -> n2 ) ( binary point is at ONE )
           ( Ø <= n1 < 1 ; 1 <= n2 < 2 )
: EXP2* ONE 2
     BEGIN DUP 2- EFACTORS DUP >R 3 PICK SWAP <
          IF R> DROP 1+
          ELSE ROT R> - -ROT 2DUP >SHIFT ROT + SWAP
          THEN DUP 13 =
     UNTIL DROP NIP
;

SCR# 53
\ Log-Point Number Words
 ( -> n1 ) ( n1 is number of digits in fraction for LSTR$ )
: #DIGITS 236Ø ( LOG2 OF 738: PRECISION LIMIT OF 16-BIT LOG #)
     256 BASE @ DUP >R LOG2 */ SPLIT SWAP Ø> R> 11 > NOT AND
     IF 1+ THEN     ( BASE <= 11 WITHIN RANGE WITH EXTRA DIGIT)
;
 ( 1# -> dn )
: FIXED DUP
     IF DUP >R LABS 16384 - SPLIT SWAP
          32 * 16 + EXP2* SWAP 13 - DUP Ø<
          IF NEGATE 1- >SHIFT 1+ 2/ Ø
          ELSE DUP IF 1 SWAP <SHIFT *D THEN
          THEN R> Ø< IF DNEGATE THEN
     ELSE Ø
     THEN
;
```

```
SCR# 54
\ Log-Point Number Words

 ( l# -> exp mant ) ( exp integer, mant rational)
: L>E DUP                    ( mant ^@ 2^#DIGITS )
     IF DUP >R ( sign) LABS 16384 - 256
          BASE @ LOG2 DUP >R */ SPLIT DUP 127 >
          IF -256 ( $FFØØ) OR THEN SWAP R> M*
          SWAP 4 / 1+ 2/ 8191 ( $1FFF) AND ( 2^13) EXP2*
          BASE @ 1 #DIGITS Ø
          DO OVER * LOOP NIP ONE */ ( ADJUST TO FS )
          1 ROT <SHIFT * R> Ø< IF NEGATE THEN
     ELSE Ø
     Ø Ø PAD DUP 1+ C@ ASCII - = DUP >R ( save sign )
     IF 1+ THEN CONVERT >R DABS FLOAT R> DUP C@ ASCII . =
     IF DUP 1+ >R Ø Ø ROT CONVERT >R FLOAT
          R> R> OVER >R SWAP - 2DUP AND

SCR# 55
\ Log-Point Number Words

 ( l# -> str )
: LSTR$ L>E >R DUP ABS Ø <# #S ROT SIGN ASCII ^ HOLD
     2DROP R@ ABS Ø #DIGITS Ø ?DO # LOOP
     ASCII . HOLD # R> SIGN #>
;

 ( l# -> )
: L. LSTR$ TYPE SPACE ;



SCR# 56
\ Log-Point Number Words
 ( X Y -> Z) ( X >= Ø)
: L^ OVER
     IF ?DUP
          IF DUP >R LABS 16384 - DUP >R ABS
               SPLIT ^2 SWAP 32 * EXP2*
               16 */ 1+ 2/ SWAP 16384 - R> Ø<
               IF 256 ROT */ ( X 256 Y)
               ELSE 256 */   ( X Y 256)
               THEN R> Ø<
               IF NEGATE THEN 16384 + LABS
          ELSE DROP [ 1 Ø FLOAT ] LITERAL
          THEN
     ELSE DROP
     THEN
;
```