BINAR™ Forth Floating Point Reference

Preliminary - Version 0.0

March 1, 1990

CONTENTS

This floating point package and its documentation are derived from: Koopman, P., Forth Floating Point and Integer Math, Mountain View Press, 1985. That book and the associated code were copyrighted and then placed into the public domain. However, the code and documentation here are substantially refined, and are considered Harris Semiconductor Proprietary Information.

1.0 Introduction

This document describes the floating point math package for the BINAR processor. This is a preliminary software release and, although the author is reasonably sure that the package works, some minor bugs and documentation shortcomings are to be expected.

In the spirit of FORTH, this package has no error checking. This greatly enhances program speed and simplicity at the cost of burdening the programmer with the responsibility for preventing math overflows, underflows, and other errors.

2.0 Getting Started

First, load the math package:

```
LOAD FLOAT.4
```

You must load the math package every time you cold-start the system. Now let's play with some math operations:

```
f# 1.  f# 3.  F/ F.
f# 123.331  f# 99.81  F+ F.
f# 10000.0  f# .00009  F* F.
f# 2.  SQRT F.
f# 3.  1/X F.
```

3.0 Examples of Floating Point Use

3.1 Sample Problem

You are working for the Forth Street Canning Company. Your boss wants to minimize the amount of metal in his cans and bring out a new product line that is cheaper to manufacture. He wants you to enumerate all possible can sizes that the plant machinery can manufacture so he can pick the ones with the least surface area. The can-making machines can make cans from 1 inch to 4 inches in radius in increments of 1/4 of an inch, and 1 inch to 5 inches in height in increments of 1/2 inch.

3.2 Solution

First, let's review some formulas for cylinders:

```
V = ( PI * ( R**2 ) ) * H
A = 2 * ( PI * ( R**2 ) ) + 2 * PI * R * H
```

where:
- V = volume of a can
- A = surface area of a can
- PI = 3.14159..
- R = radius of the base of the can
- H = height of the can

now, let's define the evaluation function we are trying to maximize:

```
X = V / A
```

where:
- X = efficiency of the can design, the higher the better

By the way, ignore the wise-guy in the back who yelled out something about "calculus". I don't happen to remember the equations for that, nor do I care to fuss with making the continuous equations fit into a quantized problem.

Now, let's implement the formulas we gave:

```
: VOLUME    ( fheight fradius -> fvolume )
   **2  F*   PI F*  ;
```

```
: AREA   ( fheight fradius -> farea )
   FDUP  **2  PI F*  F2*  F>R
   F*  PI F*  F2*   FR>  F+  ;

: EFFICIENCY   ( fvolume farea -> fefficiency )
   F/  ;
```

In these definitions, the "F" words are the floating point versions of integer operators. Also, PI is a constant that gives the trigonometric value pi, and **2 squares a floating point number.

Let's test the definitions:

```
f# 5.  f# 2.5  VOLUME F.
f# 5.  f# 2.5  AREA F.
f# 98.17478  f# 117.8097  EFFICIENCY F.
```

That done, we'll define a header word and the driving routine in a straight-forward, brute force manner:

```
: TAB   ( #columns -> )
   [ SYSTEM ] #c [ FORTH ]   @  -  0 MAX  SPACES ;

: .HEADER   ( -> )
   CR  ." HEIGHT"
   10 TAB  ." RADIUS"
   20 TAB  ." AREA"
   30 TAB  ." VOLUME"
   40 TAB  ." EFFICIENCY"  CR CR ;

: CAN   ( fheight fradius -> )
   FOVER F.
   10 TAB   FDUP F.
   20 TAB   FOVER FOVER AREA  FDUP F.  F>R
   30 TAB   VOLUME  FDUP F.
   40 TAB   FR>  EFFICIENCY  F.  CR ;

: CAN-HEIGHT   ( fheight -> )
   f# 1.0
   BEGIN
      KEY?  ABORT" BREAK..."
      FOVER FOVER  CAN      f# 0.25 F+
      FDUP  f# 4.25  F<  NOT   UNTIL
   FDROP FDROP  CR ;

: RESULTS   ( -> )
   .HEADER
   f# 1.0
```

```
BEGIN
   FDUP  CAN-HEIGHT       f# 0.5 F+
   FDUP  f# 5.5  F< NOT      UNTIL
FDROP ;
```

The word RESULTS displays the desired results the boss asked for. The boss would pick a volume he wanted to make a can for and find the highest efficiency size for that volume. For example, a 50 cubic inch can (plus or minus two cubic inches) could be 1 inch high by 4 inch radius, 1.5 by 3.25, 2.5 by 2.5, 4. by 2., or 5. by 1.75 . The best can would be the 4 inch high by 2 inch radius can with an efficiency of approximately 0.667 .

This application is I/O intensive -- most of the program's time is spent displaying results. For those of you who insist on using RESULTS as a speed benchmark against other languages, please consider the following CALCULATIONS benchmark as well. You will find that there is a tradeoff between calculation speed and I/O speed in any floating point package.

```
( Calculation speed and accuracy benchmark )
( Adapted from BYTE magazine )
(   Volume 10, No. 5, MAY 1985, page 280 )
f# 2.71828 FCONSTANT FA
f# 3.14159 FCONSTANT FB

: CALCULATIONS    ( -> )
   f# 1.0
   5000 0 DO     FA  F*   FB  F*
                 FA  F/   FB  F/   LOOP
   CR ." DONE" CR ." ERROR="  f# 1.0 F-  F.  ;
```

3.3 More Floating Point

Floating point numbers are good at representing APPROXIMATIONS of very large or small numbers. Try the following example:

```
f# 12345678912343212.  F.
f# .00000000987654321345345 4345  F.
f# 78.9  F.
```

As you can see, F. is "smart" in the sense that it automatically displays very large or small magnitude numbers in scientific notation. You can also see that only seven digits of the number are printed out, since this is an approximation technique.

If you want to display a number in fixed point notation, regardless of size, or you want to force a number to be displayed in scientific notation, use F.E or F.X as shown in the following examples:

```
f# 1.23  10 EXP   F.E  CR   f# 6.78  F.E
f# 1.23  10 EXP   F.X  CR   f# 6.78  F.X
```

The word EXP, when preceded by a single precision integer, allows you to enter an exponent for a floating point number.

For the "nickel tour" of the transcendental and exponential floating point features of this package, enter the following:

```
f# 30.  SIN F.
f# 1.0  ATAN F.
f# 20.  LOG F.
f# 1.30103  10** F.
```

There are many more transcendental and exponential functions in the package.

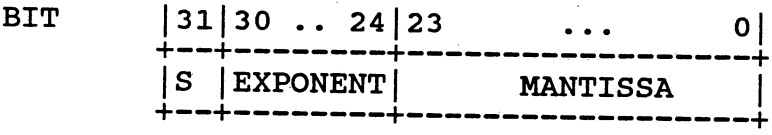
4.0 Numeric Formats

4.1 Single Precision Floating Point Numbers

Floating point numbers in this package exactly correspond to IEEE Floating Point Standard (Task P754) short floating point numbers in bit layout. However, NaN's, infinities, denormalized numbers, etc. are not supported by this implementation.

Floating point numbers range in value
from +/-5.9 * (10** -39)
to +/-6.8 * (10** 38)
with approximately six and two-thirds decimal digits of precision, plus a representation for exact zero.

The bit format of a floating point number is:



Bit 31 is the highest order bit and bit 0 is the lowest order bit. S is the sign of the mantissa with 0 meaning positive and 1 meaning negative. EXPONENT is the 8-bit radix 2 exponent with a bias of 127 decimal (subtract 127 from this 8 bit value to convert it to a two's complement integer exponent). MANTISSA is an unsigned 23-bit mantissa with an implied high-order "phantom" 1 bit. The decimal point for the mantissa is to the right of the "phantom" 1 bit. This gives a 24-bit effective mantissa but only uses 23 bits of actual storage. S and MANTISSA together form a signed-magnitude mantissa, NOT a two's complement mantissa.

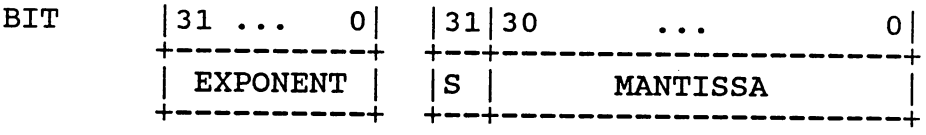
The value of a floating point number is:
(-1 * S) * (0 . 1 <MANTISSA>) * (2 ** (EXPONENT-127))

Zero is a special value represented by all bits being zero in this scheme. "Negative zero" is not a valid representation.

4.2 Temporary Floating Point Numbers

Temporary floating point numbers correspond to IEEE extended single precision floating point numbers, and are used to improve the number of significant bits for intermediate results, especially in transcendental functions. The conversion from single precision ("F" format) to temporary ("T" format) is relatively expensive, so you may want to use T format for intermediate results within a long calculation as well.

The format for a temporary floating point number is a pair of 32-bit words:



EXPONENT is a two's complement integer with the radix 2 exponent of the number. S is the sign bit of the mantissa, with 0 a positive sign and 1 a negative sign. MANTISSA is a left-normalized unsigned mantissa, with bit 30 in the MANTISSA 1 except for when the floating point number is exactly zero. S and MANTISSA form a signed magnitude mantissa. The sign bit is not forced to zero by temporary operations, so it is possible to have the number -0. Also, the exponent is not necessarily forced to zero when the mantissa is zero. All T operations that check for zero take this into account, and strip the sign bit as well as zeroing the exponent when required (this is faster overall than checking for these conditions after every operation). The exponent is placed on top of the stack, and the mantissa is kept as the second stack element when a T-number is place on the stack.

5.0 Implementation Notes

5.1 Pros and Cons of Radix 2

Radix 2 floating point numbers were chosen for this package due to their compatibility with the IEEE floating point standard and the 8087 chip. Below are some pros and cons of this approach compared with approaches using decimal encoding of some sort:

PROS:

1) The format is IEEE standard.

2) Radix 2 is well-suited to fast arithmetic. Normalization and alignment of decimal points is accomplished by bit shifting instead of multiplication and division by 10.

3) Radix 2 numbers can easily be converted to any base. While radix 10 numbers can theoretically be converted to any base, in practice the necessary routines are usually not implemented. This package is implemented so that I/O of floating point numbers can be performed in any base, including base 10.

4) The 32-bit floating point format used here can use the Forth data stack. In particular, this package's floating point numbers are identical in size to single precision integers. Most radix 10 number schemes do not use 2 or 4 or 8 byte numbers, and require special stack operations or an entirely separate floating point stack.

CONS:

1) I/O is relatively slow for radix 2 numbers, since logarithms are used to convert base 2 exponents to base 10 exponents.

5.2 Floating Point Accuracy

The 32-bit floating point numbers are accurate to about six and two-thirds decimal digits. The default value for SIGDIG is 7, so on occasion there may be slightly surprising results when printout floating point numbers. If fewer digits are desired for display in the F., F.E or F.X

formats, SIGDIG can be changed using SIGDIG!. This change will affect display only -- all calculations will still be done with all available bits of precision.

5.3 Algorithms

The transcendental approximation formulas used in this package were derived from various sources. Most are based on Chebyshev and Taylor polynomials. All polynomials used are designed to theoretically deliver at least 7 digits after the decimal point. Several "TEST" words are included at the end of the FLOAT.4 file to check these results.

Due to cumulative roundoff errors in calculations and conversions, the seventh digit of a transcendental function may, on occasion, be slightly off. If this causes a problem with displaying exact results, set SIGDIG to 6 and use the 7th digit as a guard digit to hide roundoff errors.

5.4 Notation

Each entry in the glossary includes the following information:

1) The name, stack notation before and after execution, and a designation with regard to its use.

2) The pronunciation of the ideogram.

3) The form of the use of the ideogram where appropriate.

4) The functional definition. This is a concise description of what the word does.

5) An example, which may be an important or representative definition from the source code, a reasonable use, or in a few cases a contrived use. With each example is a short discussion of the example.

6) An optional comment, which tries to put that ideogram in some sort of perspective. It may discuss some of the quirks or subtle features of the word.

7) Occasional notes have been added at points of particular concern or where caution must be used.

Standard Forth notation has been extended to include floating point and temporary floating point numbers:

flag - truth value. 0=false, non-zero=true
n - single precision integer (32 bits)
d - double precision integer (64 bits)
f - floating point number (32 bits)
t - temporary floating point number (64 bits)
un - unsigned single precision integer (32 bits)

Many word names were created by prefixing existing word names with a "stroke" to denote that the newly defined word has the same function as the old word, but works on a different size of operand. The "strokes" are:

D - double precision integer
F - floating point number
T - temporary floating point number

Thus, a family of words: D+ , F+ , and T+ are all defined as addition, but with each word requiring different types on inputs from the stack.

6.0 Glossary

****2** f1 -- f2 "squared"

Multiply f1 by itself, squaring it.

Example:
```
f# 3. **2 F.
```
Input, calculate and display 3 raised to the 2nd power, yielding 9.

1/X f1 -- f2 "one-over-x"

Computes the reciprocal f2 = 1/f1 .

Example:
```
f# 2. 1/X F.
```
0.5 is the reciprocal of 2.

10** f1 -- f2 "ten-to-the-x"

Raises 10 to the power of the input number f1, giving the result f2.

Example:
```
f# 4.7 10** F.
```
Input, compute and display 10**4.7

Comment: This is a special case of the word ** which is commonly used when working with base 10 (decimal) logarithms.

2** f1 -- f2 "f-two-to-the-x"

Raises 2 to the power of the input number f1, giving the result f2.

Example:
```
f# 4.7 2** F.
```
Input, calculate and display 2 raised to the 4.7th power

Comment: This is the primitive word used by the temporary floating point package for exponentiation in all bases.

<ACOS> f1 -- f2 "bracket-arc-cosine"

Computes f2, the angle (in radians) whose cosine is f1. f1 must be in the range of -1 to 1. f2 is returned in the range 0 to pi.

Example:
```
f# .5 <ACOS> RAD>DEG F.
```
Computes the arc-cosine of 0.5, which is 60 degrees.

Comment: This is the trigonometric inverse function to <COS>. <ACOS> is typically used for intermediate calculations to eliminate the radian to degree conversion overhead in ACOS.

<ACOT> f1 -- f2 "bracket-arc-cotangent"

Computes f2, the angle (in radians) whose cotangent is f1. The value of f1 is not restricted. f2 is returned in the range 0 to pi.

Example:
```
f# .8390996 <ACOT> RAD>DEG F.
```
Computes the arc-cotangent of .8390996, which is 50 degrees.

Comment: This is the trigonometric inverse function to <COT>. <ACOT> is typically used for intermediate calculations to eliminate the radian to degree conversion overhead in ACOT.

<ACSC> f1 -- f2 "bracket-arc-cosecant"

Computes f2, the angle (in radians) whose cosecant is f1. f1 must be in the range of 1 to infinity or -1 to -infinity. f2 is returned in the range 0 to pi/2 or -pi to -pi/2 .

Example:

```
f# 1.414214  <ACSC>  RAD>DEG  F.
```

Computes the arc-cosecant of 1.414214, which is 45 degrees.

Comment: This is the trigonometric inverse function to <CSC>. <ACSC> is typically used for intermediate calculations to eliminate the radian to degree conversion overhead in ACSC.

<ASEC> f1 -- f2 "bracket-arc-secant"

Computes f2, the angle (in radians) whose secant is f1. f1 must be in the range of 1 to infinity or -1 to -infinity. f2 is returned in the range 0 to pi/2 or -pi to -pi/2.

Example:

```
f# 1.064178  <ASEC>  RAD>DEG  F.
```

Computes the arc-secant of 1.064178, which is approximately 20 degrees.

Comment: This is the trigonometric inverse function to <SEC>. <ASEC> is typically used for intermediate calculations to eliminate the radian to degree conversion overhead in ASEC.

<ASIN> f1 -- f2 "bracket-arc-sine"

Computes f2, the angle (in radians) whose sine is f1. f1 must be in the range of -1 to 1. f2 is returned in the range -pi/2 to pi/2.

Example:

```
f# .5 <ASIN> RAD>DEG F.
```

Computes the arc-sine of 0.5, which is 30 degrees.

Comment: This is the trigonometric inverse function to <SIN>. <ASIN> is typically used for intermediate calculations to eliminate the radian to degree conversion overhead in ASIN.

<ATAN> f1 -- f2 "bracket-arc-tangent"

Computes f2, the angle (in radians) whose tangent is f1. The range of f1 is unrestricted. f2 is returned in the range -pi/2 to pi/2.

Example:

```
f# 1. <ATAN> RAD>DEG F.
```

Computes the arc-tangent of 1., which is 45 degrees.

Comment: This is the trigonometric inverse function to <TAN>. <ATAN> is typically used for intermediate calculations to eliminate the radian to degree conversion overhead in ATAN.

<ATAN2> f1 f2 -- f3 "bracket-arc-tangent-two"

Computes f3, the angle (in radians) whose tangent is (f2/f1). The ranges of f1 and f2 are unrestricted. f3 is returned in the range -pi to pi.

Example:

```
f# -1. f# 0  <ATAN2>  RAD>DEG  F.
```

Computes the arc-tangent of the x-y coordinate (-1,0), which is 180 degrees.

Comment: This is the trigonometric inverse function to <TAN> with the special ability to return a result from -pi to pi. <ATAN2> takes into account the signs of f1 and f2 and returns the result f3 in the correct quadrant if f1 is interpreted as an x coordinate and f2 is interpreted as a y coordinate on a Cartesian coordinate system. This word has obvious uses in graphics, and is used when doing rectangular to polar coordinate conversion. <ATAN2> is typically used for intermediate calculations to eliminate the radian to degree conversion overhead in ATAN2.

<COS> f1 -- f2 "bracket-cosine"

Computes f2, the cosine of the angle f1. f1 is in radians. The range of f1 is unrestricted. f2 is returned in the range -1 to 1.

Example:

```
f# 45.  DEG>RAD  <COS>  F.
```

Computes the cosine of 45 degrees.

Comment: <COS> is typically used for intermediate calculations to eliminate the degree to radian conversion overhead in COS.

<COT> f1 -- f2 "bracket-cotangent"

Computes f2, the cotangent of the angle f1. f1 is in radians. f1 must not be an even multiple of pi/2.

Example:
```
f# 60.  DEG>RAD  <COT>  F.
```
Computes the cotangent of 60 degrees.

Comment: <COT> is typically used for intermediate calculations to eliminate the degree to radian conversion overhead in COT.

<CSC> f1 -- f2 "bracket-cosecant"

Computes f2, the cosecant of the f1. f1 is in radians. f1 must not be an even multiple of pi/2

Example:
```
f# 45.  DEG>RAD  <COS>  F.
```
Computes the cosecant of 45 degrees.

Comment: <CSC> is typically used for intermediate calculations to eliminate the degree to radian conversion overhead in CSC.

<F.> d1 n2 -- addr count n3 "bracket-f-dot"

Convert double precision integer d1 representing a mantissa and exponent n2 (defined in the same manner as the outputs of the word F>ME) to a fixed point character string. addr is the address of the first character in the string, count is the number of characters in the string, and n3 is the number of characters after the embedded decimal point in the string (including trailing zeros.) If n3 is less than 1, then there are no trailing digits after the decimal point.

Example:

```
f# 12.3  F>ME  <F.>  DROP  TYPE
```

Converts the number 12.3 to a character string and prints it with trailing zeros.

Comment: The format of the output character string is: a negative sign (if the number is negative,) followed by one or more digits, followed by a decimal point, followed by enough digits after the decimal point to form SIGDIG worth of significant digits (non-leading-zero digits in the case of a number less than one.) Trailing zeros may be generated if SIGDIG is greater than the number of significant digits in the value converted to the character string. The word <F.> is used by all fixed-point number display routines.

<FNUMBER> addr -- f "bracket-f-number"

Convert the count and character string at addr to a signed 32-bit floating point number using the current base. If numeric conversion is not possible, an error condition exists. The string may contain a preceding negative sign.

Example:

```
: INPUT  ." INPUT AN INTEGER: "
         QUERY  BL WORD  <FNUMBER>  CR F.  ;
```

This definition provides for a prompt and then a pause for the operator to input the requested number. The input character stream is parsed, converted to a floating point value and displayed.

Comment: This ideogram will recognize two non-numeric characters: a decimal point and a leading negative sign. The position of the decimal point is recorded in the user variable DPL.

<P>R> fradius fangle -- fx fy "bracket-p-to-r"

Performs polar to rectangular coordinate conversion. The vector with length fradius and orientation fangle (in radians) is converted to an x,y coordinate pair (fx,fy).

Example:

```
f# 7.8  f# 30.  DEG>RAD  <P>R>  FSWAP  F. F.
```

A vector with length 7.8 and angle 30. is converted to an x,y coordinate.

Comment: <P>R> converts a polar coordinate pair (radius,angle) to rectangular coordinates (x,y). In addition to geometric calculations, <P>R> is useful for complex number manipulations.

<R>P> fx fy -- fradius fangle "bracker-r-to-p"

Performs rectangular to polar coordinate conversion. The Cartesian coordinate pair (fx,fy) is converted to a radius fradius and an angle fangle (in radians).

Example:

```
f# 3.1  f# -5.1  <R>P>  RAD>DEG  FSWAP  F. F.
```

A vector with x-value 3.1 and y-value -5.1 has a length of 5.968249 and makes and angle of -58.70696 degrees with the x-axis.

Comment: <R>P> converts a rectangular coordinate pair (x,y) to polar coordinates consisting of the vector length and an angle from the positive x axis to the vector. In addition to geometric calculations, <R>P> is useful for complex number manipulations. <R>P> uses <ATAN2> to compute fangle, so range restrictions are the same as <ATAN2>.

<SEC> f1 -- f2 "bracket-secant"

Computes f2, the secant of the angle f1. f1 is in radians. f1 must not be an odd multiple of pi/2.

Example:
```
f# 45.  DEG>RAD  <SEC>  F.
```
Computes the secant of 45 degrees.

Comment: <SEC> is typically used for intermediate calculations to eliminate the degree to radian conversion overhead in SEC.

<SIN> f1 -- f2 "bracket-sine"

Computes f2, the sine of the angle f1. f1 is in radians. The range of f1 is unrestricted. f2 is returned in the range -1 to 1.

Example:
```
f# 45.  DEG>RAD  <SIN>  F.
```
Computes the sine of 45 degrees.

Comment: <SIN> is typically used for intermediate calculations to eliminate the degree to radian conversion overhead in SIN.

<TAN> f1 -- f2 "bracket-tangent"

Computes f2, the tangent of the angle f1. f1 is in radians. f1 must not be an odd multiple of pi/2.

Example:
```
f# 60.  DEG>RAD  <TAN>  F.
```
Computes the tangent of 60 degrees.

Comment: <TAN> is typically used for intermediate calculations to eliminate the degree to radian conversion overhead in TAN.

<TATAN> t1 -- t2 "bracket-t-arc-tangent"

Computes t2, the angle (in radians) whose tangent is t1. The range of t1 is unrestricted. t2 is returned in the range -pi/2 to pi/2.

Example:

```
t# 1.  <TATAN>  TRAD>DEG  T>F  F.
```

Computes the arc-tangent of 1., which is 45 degrees.

<TCOS> t1 -- t2 "bracket-t-cosine"

Computes t2, the cosine of the angle t1. t1 is in radians. The range of t1 is unrestricted. t2 is returned in the range -1 to 1.

Example:

```
t# 45.  TDEG>RAD  <TCOS>  T>F  F.
```

Computes the cosine of 45 degrees.

<TNUMBER> addr -- t "bracket-t-number"

Convert the count and character string at addr to a signed temporary floating point number using the current base. If numeric conversion is not possible, an error condition exists. The string may contain a preceding negative sign.

Example:

```
: INPUT  ." INPUT AN INTEGER: "
         QUERY  BL WORD  <TNUMBER>  CR  T>F F.  ;
```

This definition provides for a prompt and then a pause for the operator to input the requested number. The input character stream is parsed, converted to a floating point value and displayed.

Comment: This ideogram will recognize two non-numeric characters: a decimal point and a leading negative sign. The position of the decimal point is recorded in the user variable DPL.

<TSIN> t1 -- t2 "bracket-t-sine"

Computes t2, the sine of the angle t1. t1 is in radians. The range of t1 is unrestricted. t2 is returned in the range -1 to 1.

Example:
```
t# 45. TDEG>RAD <TSIN> T>F F.
```
Computes the sine of 45 degrees.

?FNEGATE f1 n -- f2 "query-f-negate"

If n is less than zero, negate f1, otherwise do nothing. Leave the result as f2.

Example:
```
f# 22 -5 ?FNEGATE F.
```
Place 22 on the stack and apply the sign of -5 to it, giving the result of - 22.

Comment: This is a floating point version of the word ?NEGATE .

?TNEGATE t1 n -- t2 "query-t-negate"

If n is less than zero, negate t1, otherwise do nothing. Leave the result as t2.

Example:
```
t# 22 -5 ?TNEGATE T>F F.
```
Place 22 on the stack and apply the sign of -5 to it, giving the result of - 22.

Comment: This is a temporary floating point version of the word ?NEGATE .

ACOS f1 -- f2 "arc-cosine"

Computes f2, the angle (in degrees) whose cosine is f1. f1 must be in the range of -1 to 1. f2 is returned in the range 0 to 180.

Example:
```
f# .5 ACOS F.
```
Computes the arc-cosine of 0.5, which is 60 degrees.

Comment: This is the trigonometric inverse function to COS .

ACOT f1 -- f2 "arc-cotangent"

Computes f2, the angle (in degrees) whose cotangent is f1. The value of f1 is not restricted. f2 is returned in the range 0 to 180.

arc-cotangent

Example:
```
f# .8390996 ACOT F.
```
Computes the arc-cotangent of .8390996, which is 50 degrees.

Comment: This is the trigonometric inverse function to COT .

ACSC f1 -- f2 "arc-cosecant"

Computes f2, the angle (in degrees) whose cosecant is f1. f1 must be in the range of 1 to infinity or -1 to -infinity. f2 is returned in the range 0 to 90 or -180 to -90.

Example:
```
f# 1.414214 ACSC F.
```
Computes the arc-cosecant of 1.414214, which is 45 degrees.

Comment: This is the trigonometric inverse function to CSC .

ASEC f1 -- f2 "arc-secant"

Computes f2, the angle (in degrees) whose secant is f1. f1 must be in the range of 1 to infinity or -1 to -infinity. f2 is returned in the range 0 to 90 or -180 to -90.

Example:
```
f# 1.064178  ASEC   F.
```
Computes the arc-secant of 1.064178, which approximately 20 degrees.

Comment: This is the trigonometric inverse function to SEC .

ASIN f1 -- f2 "arc-sine"

Computes f2, the angle (in degrees) whose sine is f1. f1 must be in the range of -1 to 1. f2 is returned in the range -90 to 90.

Example:
```
f# .5  ASIN  F.
```
Computes the arc-sine of 0.5, which is 30 degrees.

Comment: This is the trigonometric inverse function to SIN .

ATAN f1 -- f2 "arc-tangent"

Computes f2, the angle (in degrees) whose tangent is f1. The range of f1 is unrestricted. f2 is returned in the range -90 to 90.

Example:
```
f# 1.  ATAN  F.
```
Computes the arc-tangent of 1., which is 45 degrees.

Comment: This is the trigonometric inverse function to TAN .

ATAN2 f1 f2 -- f3 "arc-tangent-two"

Computes f3, the angle (in degrees) whose tangent is (f2/f1). The ranges of f1 and f2 are unrestricted. f3 is returned in the range -180 to 180.

Example:
```
f# -1. f# 0  ATAN2  F.
```
Computes the arc-tangent of the x-y coordinate (-1,0), which is 180 degrees.

Comment: This is the trigonometric inverse function to TAN with the special ability to return a result from -180 to 180. ATAN2 takes into account the signs of f1 and f2 and returns the result f3 in the correct quadrant if f1 is interpreted as an x coordinate and f2 is interpreted as a y coordinate on a Cartesian coordinate system. This word has obvious uses in graphics, and is used when doing rectangular to polar coordinate conversion.

BASE! n -- "base-store"

Store a new value in BASE, properly updating BASE_VAL for correct operation of floating point output conversion.

Example:
```
: DECIMAL   10 BASE! ;
```

Comment: BASE! should always be used instead of BASE ! to ensure proper operation of the floating point package. HEX and DECIMAL are redefined by the package for automatic use of BASE!.

BASE_VAL -- addr "base-val"

Variable that contains BASE**SIGDIG for speeding up output numeric conversions.

Example:

```
: BASE!  ( n --- )
   DUP BASE !
   1 SIGDIG @ 0 DO OVER *  LOOP
   BASE_VAL !   DROP ;
```

Comment: This variable saves having to perform a multiply loop every time F>ME is executed. Note that DECIMAL and HEX are redefined to use BASE! instead of BASE !. Also, SIGDIG! must be used to change the number of significant digits for correct operation.

COS f1 -- f2 "cosine"

Computes f2, the cosine of the angle f1. f1 is in degrees. The range of f1 is unrestricted. f2 is returned in the range -1 to 1.

Example:

```
f# 45.  COS  F.
```

Computes the cosine of 45 degrees.

COT f1 -- f2 "cotangent"

Computes f2, the cotangent of the angle f1. f1 is in degrees. f1 must not be and even multiple of 90.

Example:

```
f# 60.  COT  F.
```

Computes the cotangent of 60 degrees.

CSC f1 -- f2 "cosecant"

Computes f2, the cosecant of the angle f1. f1 is in degrees. The range of f1 is unrestricted. f1 must not be an even multiple of 90.

Example:
```
f# 45.  COS  F.
```
Computes the cosecant of 45 degrees.

D, d -- "d-comma"

Allot eight bytes in the dictionary, storing d there in standard double precision format.

Example:
```
12. D,
```
Extends the dictionary space by eight bytes and places the number 12 in the allocated space.

Comment: This is a double precision version of the word ",".

DCONSTANT d -- "d-constant"

A defining word used to create a dictionary entry for <name>, leaving d in its parameter field. When <name> is later executed, d will be left on the stack.

Form: DCONSTANT <name>

Example:
```
33.33   DCONSTANT   NEW-VALUE
```
Enter the value 33.33, which will be a double precision number, and store it in an ideogram named NEW-VALUE. Executing NEW-VALUE will then cause the 3333 to be placed on the stack, the location of the decimal point being lost.

Comment: This is a double precision version of the word "CONSTANT".

DEG>RAD f1 -- f2 "degrees-to-radians"

Converts the angle f1, in degrees, to the angle f2, in radians.

Example:
```
     f# 180  DEG>RAD  F.
```
Pi is 180 degrees.

Comment: DEG>RAD is the inverse operation of RAD>DEG. It multiplies the input angle in degrees by (2*pi)/360 to produce the output angle in radians.

DLITERAL C, I "d-literal"
d -- d (executing)
d -- (compiling)

IMMEDIATE WORD
If compiling, compile a stack double number into a literal. Later execution of the definition containing this literal will push it to the stack. If executing, the number will remain on the stack.

DVARIABLE --

A defining word used to create a dictionary entry of <name> and assign 8 bytes for storage in the parameter field, When <name> is later executed, it will leave the address of the first byte of its parameter field on the stack.

Form: DVARIABLE <name>

Example:
```
     DVARIABLE NEW-VALUE
```
Make a new ideogram referring to a double precision variable. Its new value is not initialized.

Comment: This is a double precision version of the word VARIABLE.

E** f1 -- f2 "e-to-the-x"

Take the inverse natural log of f1.

Example:
```
f#  12.3  E**   F.
```
Input, compute and display the floating point value.

Comment: E** is the inverse function of LN.

EXP f1 n2 -- f3 "exp"

Apply the exponent n2, using the current value of BASE, to the floating point number f1, leaving the result f3.

Example:
```
f# 12.345  10 EXP  F.
```
Display the number 1.2345 EXP 11.

Comment: This is a convenient way to enter exponents for scientific notation entry. Using EXP multiplies the top floating point number by BASE raised to the power of the exponent n2. EXP may be used at any time, not necessarily immediately after an f# command. Also, the use of EXP after f# is optional, allowing input flexibility.

F! f addr -- "f-store"

Store f as an integer.

Example: CAUTION: May corrupt or crash your system.
```
f#  3333  HEX  6A75  F!  DECIMAL
```
Stores the number 333 in 4 bytes starting at memory location 6A75.

Comment: This is a floating point version of the word ! . F! is an alias of ! .

f# -- f "f-number"

IMMEDIATE WORD
Interpret the number immediately following in the input stream as a floating point number.

Example:
```
f#  123  F.
```
This forces the normally single precision number to be interpreted as a floating point 123.

Comment: This word forces input into floating point format regardless of whether the number has a decimal point. f# is "state-smart" and either leaves the number on the stack or compiles it as a literal.

F* f1 f2 -- f3 "f-times"

Leave the arithmetic floating point signed product of f1 times f2, giving f3.

Example:
```
f# 3.  f# -6.  F*  F.
```
Compute the product of 3 and -6, which is -18.

Comment: This is a floating point version of the word *.

F** f1 f2 -- f3 "f-exponential"

Computes f3 = f1 ** f2, raising f1 to the f2 power.

Example:
```
f# 3.3  f# 10.7  F** F.
```
Raise 3.3 to the 10.7th power.

Comment: F** uses logarithms to accomplish exponentiation.

F+ f1 f2 -- f3 "f-plus"

Leave the arithmetic sum of f1 plus f2 as the result f3.

Example:
```
f# 2.2  f# 3.3  F+  F.
```
Add the two numbers 2.2 and 3.3, giving a result of 5.5
.

Comment: A floating point version of the word "+".

F+! f addr -- "f-plus-store"

Add f to the 32-bit value stored at addr, by the convention given for F+.

Example: CAUTION: May corrupt or crash your system.
```
f#  2.2  30000  F+!
```
Increment the floating point number stored at memory address 30000 by 2.2.

Comment: This is a floating point version of the word +!.

F, f -- "f-comma"

Allot 4 bytes in the dictionary, storing f there in standard floating point format.

Example:
```
f# 12  F,
```
Extends the dictionary space by 4 bytes and places the number 12 in the allocated space

Comment: This is a floating point version of the word , .

F- f1 f2 -- f3 "f-minus"

Leave the arithmetic difference of f1 minus f2 as the result f3.

Example:
```
     f# 22.  f# 33.  F-  F.
```
Subtract 33 from 22, leaving -11.

Comment: A floating point version of the word - .

F. f -- "f-dot"

Display f converted according to BASE in a free-field format, with one trailing blank. Display the sign only if negative.

Example:
```
     f#  2.2  F.
```
Input and display the floating point value 2.2.

Comment: This is a floating point version of the word ".". It is "smart" in the sense that it will either use scientific notation format for very large or very small numbers and fixed point format with trailing zeros suppressed for small numbers when outputting.

F.A f1 n2 -- "f-dot-a"

Adjust the number of digits and display f1 converted according to BASE in a fixed point format with n2 digits displayed to the right of the decimal point and a single trailing blank. Display a leading negative sign if f1 is negative. Negative values of n2 are forced to zero.

Example:

```
f#  2.2  3  F.A
```

Input the floating point value and display it as 2.2000 . The value 3 for n2 forces extension of the number to three digits after the decimal point.

Comment: This word is useful for forcing a set number of digits to be displayed after the decimal point, and is similar in effect to using the "FIX" function on many calculators. The prescribed number of digits are displayed past the decimal point using truncation or right-padding with zeros as required.

F.AR f1 n2 n3 -- "f-dot-a-r"

Adjust and Right justify the display of f1 converted according to BASE in a fixed format with n2 digits displayed to the right of the decimal point, left-padded with blanks to be right justified in an n3 character field. Display a leading negative sign if f1 is negative. Negative values of n2 and n3 are forced to zero.

Example:

```
f#  2.2  3  10  F.AR
```

Input and display the value " 2.200". The value 3 for n2 forces extension of the number to 3 digits after the decimal point. The value 10 for n3 forces 5 leading blanks so that a total of 10 characters are printed.

Comment: This word is useful for forcing decimal points to line up in producing a column of right-adjusted numbers. While this format would be good for printing dollars and cents figures, DO NOT USE floating point for monetary calculations since it will introduce round-off errors.

F.E f -- "f-dot-e"

Display f converted according to BASE in a free-field scientific notation format, with one trailing blank. Display the sign if negative.

Example:
```
     f# -1452.2  F.E
```
Input and display the value "-1.452200 EXP 3", which is 145.2 in scientific notation. The mantissa is -1.4522 and the exponent is 3.

Comment: This word outputs the number in scientific notation, where the mantissa has an absolute value less than BASE, and the total value of the number may be computed by the formula: value = mantissa * (BASE ** exponent)

F.ER f1 n2 -- "f-dot-e-r"

Display f1 converted according to BASE in scientific notation left-padded with blanks to be right-justified in an n2 character field. Display the sign if negative.

Example:
```
     f# .0312  20  F.ER
```
Input and display the value "   3.120000 EXP -2", which is .0312 in scientific notation. The result is left padded with blanks so that a total of 20 characters are printed.

Comment: This word outputs the number in scientific notation, where the mantissa has an absolute value less than BASE, and the total value of the number may be computed by the formula: value = mantissa * (BASE ** exponent)

F.R f1 n2 -- "f-dot-r"

Display f1 converted according to BASE, right adjusted in an n2 character field (left-padded with blanks.) Display the sign only if negative.

Example:
```
f# 2.2  10  F.R
```
Input and display the value 2.2 with 8 preceding blanks.

Comments: This is a floating point version of the word .R . It allows formatted output for aligning numbers in columns according to the output conventions of F. .

F.X -- "f-dot-x"

Display f1 converted according to BASE in a fixed point format and a single trailing blank. Display a leading negative sign if f1 is negative.

Example:
```
f# 2.2  F.X
```
Input and display the value "2.200000" assuming that SIGDIG is 7. The number printed is right-padded with zeros to always have at least SIGDIG digits.

Comment: This word is useful for forcing fixed point output format even for very large or very small numbers, when F. would normally produce scientific notation output.

F.XR f1 n2 -- "f-dot-x-r"

Display f1 converted according to BASE in a fixed point format left-padded with blanks to be right-justified in an n2 character field. Display a leading negative sign if f1 is negative.

Example:
```
f# 2.2  10  F.XR
```
Input and display the value " 2.200000" assuming that SIGDIG is 7. The number printed is right-padded with zeros to always have at least SIGDIG digits and left-padded with blanks to total 10 characters of display.

Comment: This word is useful for forcing fixed point output format even for very large or very small numbers, and aligning the output numbers in columns.

F/ f1 f2 -- f3 "f-divide"

Leave the arithmetic signed quotient of f1 divided by f2. f3 is the quotient with the sign as determined by conventional division sign rules.

Example:
```
f# 25.  f# 3.  F/  F.
```
Divide 25 by 3, giving a quotient of 8.333333.

Comment: This is a floating point version of the word / . Underflow is not checked for, and may yield unpredictable results.

F0< f -- flag "f-zero-less"

Leave a true (non-zero) flag if f is less than zero.

Example:
```
f# 2.2  F0<  .
```
Inputs the value and leaves a false flag of 0, since 2.2 is not less than 0.

Comment: This is a floating point version of the word 0< . The test destroys the comparand.

F0= f -- flag "f-zero-equal"

Leave a true (non-zero) flag if f is zero.

Example:
```
f# 2.2  F0=  .
```
Input, leave and display a false flag of 0, since 2.2 is not equal to 0.

Comment: This is a floating point version of the word 0= . The test destroys the comparand.

F0> f -- flag "f-zero-greater"

Leave a true (non-zero) flag if f is greater than zero.

Example:
```
f# 2.2  F0>  .
```
Input, leave and display a true flag of 1, since 2.2 is greater than 0.

Comment: This is a floating point version of the word 0> . The test destroys the comparand.

F2* f1 -- f2 "f-two-mult"

Multiplies the floating point input f1 by 2 to give the output f2.

Example:
```
f# 34.1  F2*  F.
```
Input, multiply by 2 and display 68.2.

Comment: This is a high speed multiply accomplished by adding 1 to the exponent of f1; no actual multiplication takes place -- remember that the exponent is kept in radix 2.

F2/ f1 -- f2 "f-two-divide"

Divides the floating point input f1 by 2 to give the output f2.

Example:
```
f# 34.1  F2/  F.
```
Input, calculate and display 17.05

Comment: This is a high speed divide accomplished by subtracting 1 from the exponent of f1; no actual division takes place -- remember the radix of the exponent is 2.

F< f1 f2 -- flag "f-less-than"

Leave a true (non-zero) flag if f1 is less than f2.

Example:
```
f# 4.4  f# 33.   F<  .
```
Input, test and display a true flag of 1, since 4.4 is less than 33.

Comment: This is a floating point version of the word < . The test destroys both comparands.

F= f1 f2 -- flag "f-equal"

Leave a true (non-zero) flag if f1 is equal to f2.

Example:
```
f# 22.  f# 3.3  F=  .
```
Input, test, display a false flag of 0, since 22 is not equal to 3.3.

Comment: This is a floating point version of the word = . The test destroys both comparands. The user should beware of round-off error when making tests for exact equality after calculations.

F> f1 f2 -- flag "f-greater-than"

Leave a true (non-zero) flag if f1 is greater than f2.

Example:
```
f# 22. f# 3.3 F> .
```
Input, test and display a true flag of 1, since 22 is greater than 3.3.

Comment: This is a floating point version of the word > . The test destroys both comparands.

F>D f -- d "f-to-d"

Convert the floating point input number f to the double precision integer output d. If the integer portion of f is too large to fit into d, then d will contain an overflowed value.

Example:
```
f# 3 F>D D.
```
Converts a floating point 3 to a double precision 3 and display it. Note that the sign of the number is preserved.

Comment: The number is truncated towards zero in the conversion. This is the inverse operation of D>F.

F>ME f1 - d2 n3 "f-to-m-e"

Float to Mantissa and Exponent converts the floating point input number f1 to a double precision integer mantissa d2 and a single precision integer exponent n3 using the current value of BASE. The integer d2 contains SIGDIG number of significant digits with an implied decimal point after the highest order digit.

Example:

```
7 SIGDIG! f# 123.4 F>ME . D.
```

Sets the number of significant digits for output conversion to 7 and converts the number 123.4 to a mantissa 1234000 (with an implied decimal point between the 1 and the 2,) and the exponent 2. In other words, the result is 1.234000 times 10**2.

Comment: This primitive is used for all floating point output formatting to get the number into a form that can be used with double precision integer output words.

F>N f -- n "f-to-n"

Convert a floating point number to a single precision number. Overflow is not checked.

Example:

```
f# 34. F>N .
```

Convert the floating point 34 to single precision and display it.

Comment: This is the inverse operation to N>F. The sign of the operand and the correct value are both preserved if the floating point value is within the range of single precision numbers, otherwise an overflowed value will result.

F>R f -- "f-to-r"

Transfer f to the return stack. Every F>R must be balanced by a FR> in the same control structure nesting level of a colon-definition.

Example:
```
: TEST   F>R   FDROP  FR> ;
     f#  333.33  f# 444.44  TEST  F.
```

The word TEST deletes the second floating point number on the stack by temporarily placing the top stack word on the return stack. The example drops 333.33 from the stack, leaving 444.44. Note that f# is necessary with each value if you are not in f#. F>R works the same way as.

F>T f -- t "f-to-t"

Convert a floating point number to a temporary floating point number.

Example:
```
f# 34. F>T  f# 12. F>T   T+    T>F F.
```
Convert the floating point 34 and 12 to temporary precision, add them, and print out the result.

Comment: This is the inverse operation to T>F.

F? addr -- "f-query"

Display the floating point number at addr, using the format of F. .

Example:
```
4000  F?
```
Displays the floating point number in the four bytes of memory starting at address 4000.

Comment: This is a convenient substitute for the sequence F@ F. , and is a floating point version of the word ? .

F@ addr -- f "f-fetch"

Leave the contents of the four consecutive bytes starting at addr on the stack as a floating point number f.

Example:
```
     31000  F@  F.
```
Fetches the floating point number at location 31000.

Comment: This is a floating point version of the word @ . F@ and F! are complementary functions.

FABS f1 -- f2 "f-abs"

Leave the positive floating point number f2, which is the absolute value of the floating point number f1.

Example:
```
     f#  -2.2  FABS  F.
```
Input, convert and display 2.2 .

Comment: This is the floating point version of the word ABS.

FCONSTANT f -- "f-constant"

A defining word used to create a dictionary entry for <name>, leaving f in the four bytes starting at <name>'s parameter field. When <name> is later executed, f will be left on the stack.

Form: f# 7.3 FCONSTANT <name>

Example:
```
     f#  56.  FCONSTANT  NEWCONSTANT
```
Creates a floating point constant called NEWCONSTANT and assigns a value of 56 to it.

Comment: This is a floating point version of the word CONSTANT. It allots four bytes after the parameter field starting address and stores the floating point number in the same format used by F! .

FCONVERT f1 addr1 -- f2 addr2 "f-convert"

Convert to the equivalent stack number the text beginning at addr1+1 with regard to BASE. The new value is accumulated into floating point number f1, being left as f2. addr2 is the address of the first non-convertible character.

Example:
```
HEX  f# 0  6AC2  CONVERT  DECIMAL
```
Place a floating point value of 0 on the stack and then an address in memory at which the ideogram is to begin its conversion.

Comment: This is a floating point version of the word CONVERT.

FDROP f -- "f-drop"

Drop the top floating point number on the stack.

Example:
```
f#  2.  FDROP
```
Inputs and then drops the number 2. from the stack.

Comment: This is a floating point version of the word DROP. Drops the top floating point number or any other two cells from the stack. FDROP is an alias for DROP.

FDUP f1 -- f1 f1 "f-dupe"

Duplicate the top floating point number on the stack.

Example:
```
f#  1.  FDUP  F.  F.
```
Input, duplicate and display both copies of the value.

Comment: A floating point version of the word DUP. FDUP is an alias for DUP.

FLITERAL

```
I                        "f-literal"
f  --  f    ( executing )
f  --        ( compiling )
```

If compiling, compile a stack floating point number into a literal. Later execution of the definition containing this literal will push it to the stack. If executing, the number remains on the stack unchanged.

Example:

```
: FTEST   [ f#  3.  f# 2.  F-    ]
          FLITERAL  F.  ;
FTEST
```

When executed, FTEST will print out the result of 3 - 2 which is 1. The subtraction is performed at compile time, not execution time.

Comment: This is a floating point version of the word LITERAL. Using FLITERAL allows compiling floating point constants computed at compile time.

FMAX f1 f2 -- f3 "f-max"

Leave the larger of the two floating point numbers f1 and f2 as the result f3.

Example:

```
f#  2.2  f# 3.3  FMAX  F.
```

Input values, test, and display the maximum.

Comment: This is a floating point version of the word MAX.

FMIN f1 f2 -- f3 "f-min"

Leave the smaller of the two floating point numbers f1 and f2 as the result f3.

Example:

```
f#  -2.2  f# -3.3  FMIN  F.
```

Input values, test and display the minimum.

Comment: This is a floating point version of the word MIN.

FOVER f1 f2 -- f1 f2 f1 "f-over"

Leave a copy of the second floating point number on the stack f1, and place it on top of f2.

Example:

```
f#  33.44  f# 44.44  FOVER  F.  F.  F.
```

Input values, execute function, and display all values on the stack.

Comment: A floating point version of the word OVER. FOVER is an alias for OVER.

FPICK n1 -- f2 "f-pick"

Copy the contents of the n1-th floating point stack value, not counting n1 itself, to the top of the stack. An error message is issued if n1 is less than 1.

Example:

```
f#  333.3 f# 444.4 2 FPICK F. F. F.
```

Copies 333.3 to the top of the stack.

Comment: This ideogram is a floating point version of the word PICK. It assumes that all words on the stack between the number from which f2 is copied and n1 are floating point numbers. The combination 2 FPICK is equivalent to FOVER, and 1 FPICK is equivalent to FDUP. FPICK is an alias of PICK.

FR> -- f "f-r-from"

Transfer f from the return stack to the data stack. Every FR> must be balanced by a F>R in the same control structure nesting level of a colon-definition.

Example:

```
     : TEST  F>R FDROP FR> ;
f# 333.3  f# 4444. TEST F.
```

The word TEST deletes the second floating point number on the stack by temporarily placing the top stack word on the return stack. The example drops 333.3 from the stack, leaving 4444.

CAUTION: This ideogram must be used with care to avoid crashing the system. It is useful for accessing buried numbers (although FPICK and FROLL are less dangerous) and for temporarily getting numbers out of the way of stack operations until needed later. FR> is identical to R>.

FR@ -- f "f-r-fetch"

Copy f from the top two 16-bit words on the return stack.

Example:

```
     : TEST   F>R FR@ FR> ;
     f# 333.3 TEST F. F.
```

The word TEST is equivalent to an FDUP command, but is only useful for illustration of the FR@ command. The example duplicates 333.3 on the stack.

Comment: This ideogram allows access to floating point numbers placed on the stack with F>R. It does not alter the values on the return stack.

FRAC f1 -- f2

This word returns the fractional part of f1 as f2.

Example:
```
     f# -87.123  FRAC F.
he fractional part of -87.123 is -.123
```

Comment: The fractional part of the number is given the same sign as the original number, and constitutes all the digits to the right of the decimal point.

FROLL n -- "f-roll"

Extract the n-th floating point stack element to the top of the stack, not counting n itself, moving the remaining values into the vacated position. An error message is issued if n is less than 1. All values on the data stack between the n-th nunmber and the value n itself are assumed to be floating point numbers.

Example:
```
     f# 33.3  f# 444.4  f# 44.4  3 FROLL  F. F. F.
Moves 33.3 to the top of the stack.
```

Comment: A floating point version of the word ROLL. The combination 2 FROLL is identical in function to FSWAP, and 3 FROLL is identical to FROT. FROLL is an alias for the word ROLL.

FROT f1 f2 f3 -- f2 f3 f1 "f-rote"

Rotate the top three floating point values on the stack, bringing the deepest to the top of the stack.

Example:
```
     f# 11. f# 22. f# 33.  FROT F. F. F.
Rotates the 11 to the top of the stack and prints the
numbers out in the order 11-33-22.
```

Comment: This is a floating point version of the word ROT. FROT is an alias for the word ROT.

FSWAP f1 f2 -- f2 f1 "f-swap"

Exchange the top two floating point numbers f1 and f2 on the stack.

Example:
```
f#  324. f# 523.432   FSWAP F. F.
```
Put two floating point numbers on the stack and swap their order.

Comment: This is a floating point version of the word SWAP. FSWAP is an alias for the word SWAP.

FVARIABLE -- "f-variable"

Usage: FVARIABLE <name>

A defining word used to create a dictionary entry <name> and assign four bytes for storage in the parameter field. When <name> is later executed, it will leave the address of the first byte of its parameter field on the stack.

Example:
```
FVARIABLE NEW-VALUE
f# 123.34  NEW-VALUE F!
NEW-VALUE F@ F.
```
This create a floating point variable called NEW-VALUE.

Comment: This is a floating point version of the word VARIABLE. The initial contents of the variable are unspecified. FVARIABLE is an alias for VARIABLE.

INT f1 -- f2

Truncate f1 to the next lower integer, giving f2.

Example:
```
f#  5.8227  INT   F.
```
Input, truncate and display the input down to 5.

Comment: INT is used to extract the integer portion of a number. Numbers are truncated towards zero.

LN f1 -- f2 "natural log"

Computes the natural log of f1, leaving the result f2.

Example:
```
f# 5.1  LN  F.
```
Computes the natural log of 5.1

Comment: This ideogram takes the base e log of a number. It is the inverse operation of E** .

LOG f1 -- f2

Computes the common log (base 10) of f1, leaving the result f2.

Example:
```
f# 5.1 FLOG F.
```
Computes the common log of 5.1 .

Comment: Decimal logarithms are the most commonly used logs. This is the inverse operation of 10** .

LOG2 f1 -- f2 "log-two"

Computes the log (base 2) of f1, leaving the result f2.

Example:
```
f# 5.1  LOG2 F.
```
Computes the log base 2 of 5.1 .

Comment: This is a faster logarithm than other base logs, since the floating point packaged uses base 2 logs for internal calculations. This is the inverse operation of 2** .

LOGB f1 f2 -- f3 "log-b"

Computes the log (base f2) of f1, leaving the result f3.

Example:
```
f# 5.1  f# 7.  LOGB F.
```
Computes the log base 7 of 5.1 .

Comment: This is a generalized logarithm to any base. It is the inverse operation of ** .

N>F n -- f "n-to-f"

Convert a single precision integer to a floating point number.

Example:
```
34  N>F   F.
```
Convert the integer 34 to floating point and display it.

Comment: This is the inverse operation to F>N.

N>T n -- t "n-to-t"

Convert a single precision integer to a temporary floating point number.

Example:
```
34  N>T   T>F F.
```
Convert the integer 34 to temporary floating point and display it.

Comment: This is the inverse operation to T>N.

P>R fradius fangle > fx fy "p-to-r"

Performs polar to rectangular coordinate conversion. The vector with length fradius and orientation fangle (in degrees) is converted to an x,y coordinate pair (fx,fy).

Example:
```
f# 7.8  f# 30.  P>R  FSWAP  F. F.
```
A vector with length 7.8 and angle 30. is converted to an x,y coordinate.

Comment: P>R converts a polar coordinate pair (radius,angle) to rectangular coordinates (x,y). In addition to geometric calculations, P>R is useful for complex number manipulations. P>R is the inverse operation of R>P.

PI -- f1 "pie"

A constant that leaves the floating point value of the trigonometric constant pi on the stack as f1.

Example:
```
PI F.
```

R>P fx fy -- fradius fangle "r-to-p"

Performs rectangular to polar coordinate conversion. The Cartesian coordinate pair (fx,fy) is converted to a radius fradius and an angle fangle (in degrees).

Example:
```
f# 3.1 f# -5.1  R>P  FSWAP  F. F.
```
A vector with x-value 3.1 and y-value -5.1 has a length of 5.968249 and makes and angle of -58.70696 degrees with the x-axis.

Comment: R>P converts a rectangular coordinate pair (x,y) to polar coordinates consisting of the vector length and an angle from the positive x axis to the vector. In addition to geometric calculations, R>P is useful for complex number manipulations. R>P uses ATAN2 to get the polar angle, so range restrictions are the same as for ATAN2.

RAD>DEG f1 -- f2 "radians-to-degrees"

Converts the angle f1, in radians, to the angle f2, in degrees.

Example:
```
     PI  RAD>DEG  F.
Pi is 180 degrees.
```

Comment: RAD>DEG is the inverse operation of DEG>RAD. It multiplies the input angle in radians by 360/(2*pi) to produce the output angle in degrees.

REM f1 f2 -- f3

Computes the remainder f3 when f1 is divided by f2. The remainder has the same sign as f1.

Example:
```
     f#  4.124  f# 3.  REM  F.
The remainder of 4.124/3 is 1.124
```

Comment: REM is an exact result of the floating point remainder after division. It is very useful for argument range reduction for trigonometric and other functions.

ROOT f1 f2 -- f3

Computes f3 = f1 ** (1/f2), taking the f2'nd root of f1. The sign of f1 is stripped to force it to a positive value.

Example:
```
     f#  3.12  f# 10.6  ROOT F.
Take the 10.6th root of 3.12 .
```

Comment: ROOT uses logarithms to accomplish its function. Note that SQRT is much more efficient for taking the 2nd root of a number.

SEC f1 -- f2 "secant"

Computes f2, the secant of the angle f1. f1 is in degrees. f1 must not be an odd multiple of 90.

Example:
```
f# 45.  SEC  F.
```
Computes the secant of 45 degrees.

SIGDIG -- n "sig-dig"

SIGDIG is a variable that contains the number of significant digits to be made available for displaying floating point numbers.

```
5 SIGDIG!  PI F.
7 SIGDIG!  PI F.
```
Example: Use five significant digits for floating point output operations (used by F>ME for numeric conversion.)

Comment: SIGDIG! must be used to change the number of significant digits for correct operation. The sequence SIGDIG ! does not properly set the variable BASE_VAL. Floating point calculations are always carried out with all available precision; SIGDIG only affects output operations. SIGDIG should not be set too high, or false precision will be displayed. For base 10, SIGDIG should be set to 7 or less.

SIN f1 -- f2 "sine"

Computes f2, the sine of f1 (in degrees.) The range of f1 is unrestricted. f2 is returned in the range -1 to 1.

Example:
```
f# 45.  SIN  F.
```
Computes the sine of 45 degrees.

SQRT f1 -- f2 "square-root"

Computes the square root of the absolute value of f1, giving f2.

Example:
```
f# 2.   SQRT   F.
```
Compute and display the square root of 2.

Comment: SQRT uses the Newton-Rhapson method for a higher speed square root than would be attainable using the ROOT function.

t# -- t "t-number"

IMMEDIATE WORD
Interpret the number immediately following in the input stream as a temporary floating point number.

Example:
```
t#  123  T>F  F.
```
This forces the normally single precision number to be interpreted as a floating point 123.

Comment: This word forces input into floating point format regardless of whether the number has a decimal point. t# is "state-smart" and either leaves the number on the stack or compiles it as a literal.

T* t1 t2 -- t3 "t-times"

Leave the arithmetic signed product of t1 times t2, giving t3.

Example:
```
t# 3.  t# -6.  T* T>F F.
```
Compute the product of 3 and -6, which is -18.

Comment: This is a temporary floating point version of the word * .

T+ t1 t2 -- t3 "t-plus"

Leave the arithmetic sum of t1 plus t2 as the result t3.

Example:
```
t# 2.2  t# 3.3  T+  T>F F.
```
Add the two numbers 2.2 and 3.3, giving a result of 5.5 .

Comment: A temporary floating point version of the word + .

T- t1 t2 -- t3 "t-minus"

Leave the arithmetic difference of t1 minus t2 as the result t3.

Example:
```
t# 22.  t# 33.  T-  T>F  F.
```
Subtract 33 from 22, leaving -11.

Comment: A temporary floating point version of the word - .

T/ t1 t2 -- t3 "t-divide"

Leave the arithmetic signed quotient of t1 divided by t2. t3 is the quotient with the sign as determined by conventional division sign rules.

Example:
```
t# 25.  t# 3.  T/ T>F  F.
```
Divide 25 by 3, giving a quotient of 8.333333.

Comment: This is a temporary floating point version of the word / . Underflow is not checked for, and may yield unpredictable results.

T2** t1 -- t2 "t-two-to-the-x"

Raises 2 to the power of the input number t1, giving the result t2.

Example:
```
t#  4.7  2**  T>F F.
```
Input, calculate and display 2 raised to the 4.7th power

Comment: This is the primitive word used by the floating point package for exponentiation in all bases.

T>F t -- f "t-to-f"

Convert a temporary floating point number to a floating point number. Force a "clean zero" during the conversion.

Example:
```
f# 34. F>T  f# 12. F>T   T+    T>F F.
```
Convert the floating point 34 and 12 to temporary precision, add them, and print out the result.

Comment: This is the inverse operation to F>T. Most "T" operations allow the propagation of negative zero, so T>F checks for negative zero and forces it to positive zero.

T>N t -- n "t-to-n"

Convert a temporary floating point number to a single precision number. Overflow is not checked.

Example:
```
t#  34.8  T>N  .
```
Convert the temporary floating point 34.8 to single precision 34 and display it.

Comment: This is the inverse operation to N>T. The sign of the operand and the correct value are both preserved if the temporary floating point value is within the range of single precision numbers, otherwise an overflowed value will result.

T>RND t -- n "t-to-n"

Convert a temporary floating point number to a single precision number, rounding to the nearest integer. Overflow is not checked.

Example:
```
t#  34.7  T>RND  .
```
Convert the temporary floating point 34.7 to single precision 35 and display it.

Comment: This is similar to T>N, but performs round-to-nearest conversion instead of truncation.

TABS t1 -- t2 "t-abs"

Leave the positive temporary floating point number t2, which is the absolute value of the temporary floating point number t1.

Example:
```
t#  -2.2  TABS  T>F  F.
```
Input, convert and display 2.2 .

Comment: This is the temporary floating point version of the word ABS.

TAN f1 -- f2 "tangent"

Computes f2, the tangent of f1 (in degrees.) f1 must not be an odd multiple of 90.

Example:
```
f# 60.  TAN  F.
```
Computes the tangent of 60 degrees.

TCONVERT t1 addr1 -- t2 addr2 "t-convert"

Convert to the equivalent stack number the text beginning at addr1+1 with regard to BASE. The new value is accumulated into temporary floating point number t1, being left as t2. addr2 is the address of the first non-convertible character.

Example:

```
HEX  t# 0  6AC2  CONVERT  DECIMAL
```

Place a floating point value of 0 on the stack and then an address in memory at which the ideogram is to begin its conversion.

Comment: This is a temporary floating point version of the word CONVERT.

TDEG>RAD t1 -- t2 "t-degrees-to-radians"

Converts the angle t1, in degrees, to the angle t2, in radians.

Example:

```
t# 180  TDEG>RAD  F>T  F.
```

Pi is 180 degrees.

Comment: TDEG>RAD is the inverse operation of TRAD>DEG. It multiplies the input angle in degrees by (2*pi)/360 to produce the output angle in radians.

TFRAC t1 -- t2 "t-frac"

This word returns the fractional part of t1 as t2.

Example:

```
t# -87.123  TFRAC    T>F F.
```

he fractional part of -87.123 is -.123

Comment: The fractional part of the number is given the same sign as the original number, and constitutes all the digits to the right of the decimal point.

TLOG2 t1 -- t2 "t-log-two"

Computes the log (base 2) of t1, leaving the result t2.

Example:
```
     t# 5.1  TLOG2   T>F  F.
```
Computes the log base 2 of 5.1 .

Comment: This is a faster logarithm than other base logs, since the temporary floating point packaged uses base 2 logs for internal calculations. This is the inverse operation of 2** .

TLOGB t1 t2 -- t3 "t-log-b"

Computes the log (base t2) of t1, leaving the result t3.

Example:
```
     t# 5.1  t# 7.  TLOGB    T>F F.
```
Computes the log base 7 of 5.1 .

Comment: This is a generalized logarithm to any base. It is the inverse operation of ** .

TNEGATE t1 -- t2 "t-negate"

Leave the two's complement of floating point number t1, giving t2.

Example:
```
     t#  2.2  TNEGATE  T>F F.
```
Input, negate and display a floating point number.

Comment: This is a temporary floating point version of the word NEGATE. Negative zero is allowed.

TRAD>DEG t1 -- t2 "t-radians-to-degrees"

Converts the angle t1, in radians, to the angle t2, in degrees.

Example:

```
     PI F>T  TRAD>DEG  T>F  F.
Pi is 180 degrees.
```

Comment: TRAD>DEG is the inverse operation of TDEG>RAD. It multiplies the input angle in radians by 360/(2*pi) to produce the output angle in degrees.