Chapter 9
Floating Point Arithmetic

Dave Sirag has graciously donated his implementation of the Forth Vendors Group Standard for use with JForth. This standard supports numerous arithmetic and comparison operators and also provides very flexible numeric conversion tools. Dave has also added numerous extensions that we think you will find helpful. We at Delta Research are very grateful to Dave for his contribution.

There are two main files for the floating point code, JFLT:FLOAT.FFP and JFLT:FLOAT.DOUBLE. The first file supports 32 bit single precision floating point using the Motorola Fast Floating Point Library. The second file supports 64 bit double precision IEEF format floating point for when extreme accuracy is more important than speed. The command sets between the two are essentially identical making it possible to write utilities that will work with either precision.

Without further ado, let's load the 32 bit code and try it out.

Floating Point Tutorial

First we must load the appropriate Floating Point package. I suggest starting with the single precision. If you are asked for the workbench disk please insert it as requested.

```
INCLUDE? F* JFLT:FLOAT.FFP
```

If you see a message about words not being compiled INLINE, do not worry. This just means that they could notbe compiled in the fastest possible form. It is not harmful. See INLINE in the glossary.

Now we *must* initialize this system before using it. Enter:

```
FPINIT
```

That will open up the appropriate libraries and install the proper floating point number conversion routines. Now whenever we enter a number with a decimal point in it, it will be automatically converted to floating point.

Simple Arithmetic and Output

Let's try entering some numbers and doing some simple arithmetic. Most of the arithmetic operators that Forth has for integers, "* + - / ABS MIN", etc. have their floating point counterparts. To add two floating point numbers together we simply call F+ .

```
23.5 F.  ( print a fp number )
20.0 7.55 F+ F.  ( add two fp numbers )
17.98 12.345 F/ F.  ( divide two fp numbers )
10.0 PI F* F.
```

We can also enter and display numbers in exponential notation.

```
23.7E15 E.
7.193E-8 ENG.  ( engineering format )
```

Engineering format always adjusts the exponent to a multiple of three. This is convenient for displaying seconds, milliseconds, and microseconds, or meters and kilometers, etc.

We can control the display of our numbers by using the ".R" words. If we want to display PI with 4 places after the decimal point in a field 12 characters wide, we can enter:

```
PI 4 12 F.R
8.98 99876.3 F/  2 10 ENG.R
```

Transcendental Functions

Scientific calculations often require transcendental functions like sine and cosine. This package assumes angles to be in Radians instead of degrees. Radians are a special unit of angular measurement where 360 degrees (a circle) is two Pi, or roughly 6.3, radians. Let's do some calculations. (Don't worry! If you don't understand this stuff then you probably don't need it.)

```
PI  FSIN F.  ( should be zero )
45.0 DEG>RAD FCOS F.  ( approx 0.7 )
```

Notice the use of DEG>RAD to convert between degrees and radians. RAD>DEG can be used to go the other way.

```
2.1  5.3 F** F.  ( raise 2.1 to the power of 5.3 )
137.2 FLOG F.    ( log base 10 )
13.72 FLOG F.
```

Let's calculate the hypotenuse of a triangle. Pythagoras' theorem tells us that the length of the hypotenuse of a right triangle is the square root of the sum of the squares of the two sides.

```
: HYPOT  ( a b -- c , C = SQRT( A**2 + B**2 )
    FDUP F*  ( square B )
    FSWAP FDUP F* F+ ( square A and add )
    FSQRT  ( take square root )
;
3.0 4.0 HYPOT F.  ( yep, the old 3,4,5 right triangle)
9.8 17.5 HYPOT F.
```

Precision Independent Style

Suppose you write an application that works for 32 bit single precision numbers. Wouldn't it be nice if it also worked for double precision without having to change it. By carefully using the special stack operators and storage words that Dave has written, you can do just that. The tricky part is that 32 bit numbers occupy one cell on the stack and 64 bit numbers occupy two cells. If you have two 32 bit numbers on the stack and you want to swap them you can just use SWAP . If you have two 64 bit numbers you have to use 2SWAP . If you have mixed integer and floating point things get even trickier. Luckily most of the common stack operations have been provided in a form that works no matter what precision we are using. Enter:

```
43  29.712 NFSWAP . F.
```

This version of SWAP assumes that the stack has an integer, N, and a float, F. It is defined differently for different width floating point numbers. For single precision it is just a normal SWAP , for double precision it uses ROT to do its work.

We also need special words for VARIABLE , ! and other width-dependent words. Enter:

```
FVARIABLE FVAR-1
23.4 FVAR-1 F!
FVAR-1 F@ F.
```

That will work for single or double precision. Now, suppose you want an array of numbers.

```
20 FARRAY MY-FAR  ( declare array of 20 numbers )
( store 9.876 into fourth location in array )
9.876 4 MY-FAR F!
```

Cloning Floating Point Code

Please remember that you must initialize the system before using it. Otherwise it will crash dramatically. Here is an example of a floating point program that will clone.

```
: SHOWSINES  ( -- , display sines )
```

```
        FPINIT  ( Important!! ) CR
        91 0
        DO  I DUP 4 .R  ( show angle )
            FLOAT DEG>RAD ( convert i to radians )
            FSIN   4 9 F.R CR
        LOOP
        FPTERM
    ;
```

Floating Point Glossary

This glossary has been organized by function to make it easier to find the right word.

Floating Point Control

FPINIT (--- , initialize floating point)

This *must* be called before using any floating point words or you will crash. It opens the appropriate libraries and initializes some variables. The floating point files have AUTO.INIT words that will automatically call FPINIT if they are loaded permanently in a dictionary.

FPTERM (--- , close libraries and cleanup)

Arithmetic Operators

These operators are similar to the corresponding integer operators and, therefore, don't need much explanation.

F+ (r1 r2 --- r1+r2)

F- (r1 r2 --- r1-r2)

F* (r1 r2 --- r1*r2)

F/ (r1 r2 --- r1/r2)

F2* (r --- r*2.0)

F2/ (r --- r/2,0)

FABS (r --- |r| , take absolute value of R)

FNEGATE (r --- -r)

FMAX (r1 r2 --- r1 | r2 , leave largest of r1 and r2)

FMIN (r1 r2 --- r1 | r2 , leave smallest of r1 and r2)

Result Flags

The arithmetic operators above set a variable called FPSTAT to the value in the condition code registers after the operation. These words examine FPSTAT and leave a flag if a given condition is true. They can be used to determine if an overflow had occurred or if the result was zero, etc. An overflow is what a happens when a calculation results in a number that is too big or too small for the machine.

```
FEQ  ( -- flag , true if result zero )
FLE  ( -- flag , true if result less than or equal to zero )
FLT  ( -- flag , true if result less than zero )
FGE  ( -- flag , true if result greater than or equal to zero )
FGT  ( -- flag , true if result greater than zero )
FNE  ( -- flag , true if result not equal zero )
FVC  ( -- flag , true if result did NOT overflowed )
FVS  ( -- flag , true if result overflowed )
```

Transcendental Functions

Angles are passed in radians. Use DEG>RAD or RAD>DEG to convert if needed.

```
DEG>RAD     ( r.deg -- r.rad , convert degrees to radians )
DEG/RAD     ( --- r , constant number of degrees per radian )
F**         ( r1 r2 --- r1**r2 , R1 to R2th power )
FSQRT ( r --- r**0.5 , take square root )
FLN         ( r --- ln[r] , natural logarithm )
FLOG  ( r --- log[r] , base 10 logarithm )
FALOG ( r --- 10**r , inverse logarithm )
FALN  ( r --- e**r , inverse natural log )
FSIN  ( r.rad --- sin[r] , take sine of R )
FCOS  ( r.rad --- cos[r] )
FTAN  ( r.rad --- tan[r] )
FASIN ( sin[r] --- r.rad , take arcsine of R )
FACOS ( cos[r] --- r.rad )
FATAN ( tan[r] --- r.rad )
FSINH ( r.rad --- sinh[r] , take hypersine of R )
FCOSH ( r.rad --- cosh[r] )
FTANH ( r.rad --- tanh[r] )
FCS         ( r.rad --- sin[r] cos[r] , calc both quickly )
FATCS ( sin[r] cos[r]--- r.rad , four quadrant arctangent )
```

This is similar to the ATAN2 function in FORTRAN

```
PI          ( -- pi , constant )
PI/2  ( -- pi/2 , constant )
2PI         ( -- 2pi , constant )
```

```
RAD>DEG     ( r.rad -- r.deg , convert radians to degrees )
```

Logical Operators

These operators are just like their integer counterparts except they accept floating point numbers. The following words accept two numbers.

F= F< F> F<= F>= F<> (r1 r2 -- flag)

The following words compare a number to zero.

F0= F0< F0> F0<> F0<= F0>= (r -- flag)

Stack Operators

Since floating point numbers may be 32 bits or 64 bits, we need a way to manipulate them on the stack without knowing their size. Then we can write programs that will work with either precision.

```
F>R         ( r --- , push to return stack )
FCELL+      ( n --- n' , add width of floating point number )
```

This word will add the width of a floating point cell. For single precision this would be 4+. For double precision would be 8 +.

```
FCELL-      ( n --- n' )
FCELLS      ( n --- n' )
FCELL/      ( n --- n' )
FCELLU/     ( n --- n' )
FDUP  ( r -- r r )
FNOVER      ( r n --- r n r )
FNSWAP      ( r n --- n r )
FFNROT      ( r1 r2 n --- r2 n r1 )
FNFROT      ( r1 n r2 --- n r2 r1 )
FNNROT      ( r n1 n2 --- n1 n2 r )
FOVER ( r1 r2 -- r1 r2 r1 )
FR>         ( --- r , pop from return stack )
FROT  ( r1 r2 r3 --- r2 r3 r1 )
FSWAP ( r1 r2 -- r2 r1 )
NFOVER      ( n r --- n r n )
NFFROT      ( n r1 r2 --- r1 r2 n )
NFNROT      ( n1 r n2 --- r n2 n1 )
NFSWAP      ( n r --- r n )
NNFROT      ( n1 n2 r --- n2 r n1 )
```

Number Storage

These words provide a precision-independent way of storing floating point numbers in memory and retrieving them.

```
F!          ( r addr --- , store in memory )
F@          ( addr --- r )
FARRAY      ( n <name> --- , declare an array with N cells )
FCONSTANT ( value <name> --- , declare a big enough constant )
FVARIABLE  ( <name> --- , declare a big enough variable )
```

Number Conversion Operators

For the following set of words, N is 32 bit for both double and single precision.

```
FIX         ( r --- n , round and convert to integer )
FLOAT ( n --- r , convert integer to float )
INT         ( r --- n , truncate and convert to integer )
PACK  ( d n --- r )
UNPACK      ( r --- d n )
```

Display Operators

The single precision display words support 7 significant figures. Thus 1.2345678 F. will display 1.234568 .

```
E.    ( r --- , display floating-point in exponential form )
E.R   ( r places width --- , exp form display of  R )
ENG.  ( r --- , display R in engineering exponential form )
ENG.R ( r places width --- , formatted engineering display )
F>ENGTEXT  ( r --- addr count , converts fp to eng-form text )
F>ETEXT     ( r --- addr count , converts fp to e-form text )
F>TEXT      ( r --- addr count , converts fp to text )
F.          ( r --- , display floating-point in decimal form )
F.R         ( r places width -- , formatted decimal display of R)
PLACES      ( n --- , sets default number of fractional digits )
```

Display Operators & Variables

These variables can be used to control the display of numbers. They are extensions to the FVG84 standard.

DP-CHARS (--- addr , symbols for decimal point and comma)

Allows other styles of display. The characters in this are packed as two 16 bit numbers. Here is an example of changing the decimal point and the commas.

```
COMMAS
2.345,67 F.  ( normal )
ASCII ^ DP-CHARS W!
ASCII - DP-CHARS 2+ W!
2.345,67 F.
```

E.PLUS (--- addr , if true, display "E+01" rather than "E01")

EFFLD (--- addr , width of fractional field for E. and ENG)

EXPSYMBOL (--- addr , char for "E" for E. and ENG.)

F#BYTES (-- #bytes , per float = 4 for single, 8 for double)

F.ENDPOINT (--- addr , if true, use a point at end of F.)

FFLD (--- addr , - width of fractional field for F.)

This determines the number of places after the decimal point. Range is 0 to 7 for single precision.

FLD (--- addr , total width of field for number displays)

F.EXMAX (--- addr , maximum exponent for decimal F.)

If the exponent is larger than this it will use the E. format.

FPWARN (--- addr , if true, display fp warning messages)

Number Interpreters

Floating point numbers will always be converted using base 10 (decimal) regardless of the value of BASE. This means no HEXADECIMAL floating point numbers. These determine how numbers will be input.

FLOAT.INTERPRET (--- ,allows integer decimal, and "E" form input)

FASTFP.INTERPRET (--- , allows integer and decimal form input)

FIX.INTERPRET (--- , integer input only, '.' implies double)

FNUMBER? ($string -- r true | false)

Converts a string to a floating point number and true if valid. Otherwise returns false.

NTYPE (--- addr , type of last number converted)

This will be set when a number is input. The values are 1 = int, 2 = fp, 0 = not number.