Floating-Point Library V1.2 for TurboForth V1.2

TurboForth Words

by

Lee Stewart

Table of Contents

1 Introduction

The floating-point library presented herein adds full floating-point functionality to TurboForth version 1.2.

The library itself consists of 56 TurboForth words, most of which link to a floating-point math library (FPLTF) written in TMS9900 assembler and adapted by Lee Stewart from the MDOS L10 Math Library to run on the TI-99/4A. The FPLTF is documented in the *Floating-Point Library V1.2 for TurboForth V1.2: Low-Level Support Functions Reference Guide*. It occupies 5652 bytes starting at 2000h in CPU RAM space. The TurboForth words occupy (at the time of writing) 2646 bytes of TurboForth dictionary space when all the words are loaded. A subset of words can be loaded in memory-constrained situations.

The words in the library are grouped into the following sections for discussion purposes:

- Floating-point stack manipulation words;
- Floating-point math words;
- Floating-point comparison words;
- Floating-point literal handling;
- Floating-point variables;
- Floating-point constants;
- Floating-point values;
- Displaying floating-point numbers;
- Floating-point number conversion;
- Transcendental constants and conversion functions;
- Transcendental functions.

The source code for the library is presented at the end of this document. The library itself is written in high-level Forth code, with the exception of some CODE words which handle calls to the FPLTF.

2 Acknowledgements

The author would like to thank the following for their help with the development of this library:

- Tim Tesch for the source code of the MDOS L10 Floating Point Library (FPL) and encouragement and permission in this effort to adapt it to do the heavy lifting for the TurboForth Floating Point Library.
- Beery Miller for permission to use the MDOS L10 FPL and to quote his *GenRef v1.00 MDOS Reference Guide: Math Library*.
- 9640News and all of the MDOS contributors who developed the TMS9900 code for the MDOS L10 FPL, which we adapted for use with TurboForth on the TI-99/4A.
- Mark Wills for developing the original TurboForth floating-point library, upon which some aspects of this library are based, and for developing the tagged object code loader for Turboforth, which made development of FPLTF less painful.

3 Removing the Floating-Point Library

When the floating-point library is no longer needed, it may be removed to free up the 8+ KB it occupies by **FORGET**ting the first FPL word defined in the dictionary followed by resetting low memory:

```
FORGET FPSTAT
$2000 FFAILM !
```

This restores the TurboForth environment to its state prior to loading the floating-point library.

4 Notes on Stack Signatures

Standard Forth stack signatures are used in this document to document the effects on the stack while using the floating point words. Each stack signature is enclosed in parentheses. The floating point stack is indicated with "F: " preceding the actual stack effects, whereas the data stack is not so adorned. For example:

```
S>FP ( n -- ) ( F: -- n )
```

Indicates that **S>FP** takes an integer, *n*, from the *data stack* and converts it to a *floating-point* value, *n*, which it places on the *floating-point stack*. Note that floating-point values have their *own* stack, separate from the 'standard' (integer only) stack (the data stack) built into TurboForth.

Stack effects are shown only for the affected stacks. For example:

```
FSWAP ( F: x y -- y x )
```

FSWAP does not show the data-stack effects because only the values on the floating-point stack are changed. In this case they are swapped.

5 Notes on Radix 100 Notation

TurboForth floating-point math routines use radix-100 format for floating-point numbers. The term "radix" is used in mathematics to mean "number base". We will use "radix 100" to describe the base-100 or centimal number system and "radix 10" to describe the base-10 or decimal number system. Radix-100 format is the same format used by the XML and GPL routines in the TI-99/4A console. Each floating-point number is stored in 8 bytes (4 cells) with a sign bit, a 7-bit, excess-64 (64-biased) integer exponent of the radix (100) and a normalized, 7-digit (1 radix-100 digit/byte) significand for a total of 8 bytes per floating point number. The signed, radix-100 exponent can be -64 to +63. (Keep in mind that the exponent is for radix-100 notation. Those same exponents radix 10 would be -128 to +126.) The exponent is stored in the most significant byte (MSB) biased by 64, *i.e.*, 64 is added to the actual exponent prior to storing, *i.e.*, -64 to +63 is stored as 0 to 127.

The significand (significant digits of the number) must be normalized, *i.e.*, if the number being represented is not zero, the MSB of the significand must always contain the first non-zero (significant) radix-100 digit, with the radix exponent of such a value that the radix point immediately follows the first digit. This is essentially scientific notation for radix 100. Each byte contains one radix-100 digit of the number, which, of course, means that each byte can have a value from 0 to 99 (0 to 63h) except for the first byte of a non-zero number, which must be 1 to 99. It is

easy to view a radix-100 number as a radix-10 number by representing the radix-100 digits as pairs of radix-10 digits because radix 100 is the square of radix 10. In the following list of largest and smallest possible 8-byte floating point numbers, the radix-100 representation is on the left with spaces between pairs of radix-100 digits. The radix-16 (hexadecimal) internal representation of each byte of the number is also shown:

- Largest positive floating point number [hexadecimal: 7F 63 63 63 63 63 63]:

$$99.99\,99\,99\,99\,99\,99 \times 100^{63} = 99.999999999999 \times 10^{126}$$
$$= 9.9999999999999 \times 10^{127}$$

- Largest negative floating point number [hexadecimal: 80 9D 63 63 63 63 63]:

$$-99.99\,99\,99\,99\,99\,99 \times 100^{63} = -99.999999999999 \times 10^{126}$$
$$= -9.9999999999999 \times 10^{127}$$

- Smallest positive floating point number [hexadecimal: 00 01 00 00 00 00 00 00]:

$$01.00\,00\,00\,00\,00\,00 \times 100^{-64} = 1.000000000000 \times 10^{-128}$$

- Smallest negative floating point number [hexadecimal: FF FF 00 00 00 00 00 00]:

$$-01.00\,00\,00\,00\,00\,00 \times 100^{-64} = -1.000000000000 \times 10^{-128}$$

The only difference in the internal storage of positive and negative floating point numbers is that only the first word (2 bytes) of negative numbers is negated or complemented (two's complement).

A floating point zero is represented by zeroing only the first word. The remainder of the floating point number does not need to be zeroed for the number to be treated as zero for all floating point calculations.

6 The Floating-Point Stack

The library implements a separate floating-point stack. That is, floating-point numbers are stored on their own stack, separate from normal 16-bit integer values. Pushing and popping of floating-point values is handled automatically. The size of the floating-point stack is currently set to 10, *i.e.*, there is room for 10 floating-point values. The floating-point stack size can be adjusted by changing the value assigned to the constant **`_fpssz`** in the code.

7 Entering Floating-Point Numbers in TurboForth

The number parser built into TurboForth cannot parse floating-point numbers. To specify a floating-point number, either in code, or at the command-line, simply precede the number with the word **`>F`** ("to floating-point"):

```
>F 99
>F 99.987
>F 3.14159267
>F 4.99127E-33
```

Each of the above examples, if entered directly at the keyboard, will push the above floating-point

values to the floating-point stack.

>F is a 'state-smart' word. If used inside a colon-definition like this,

```
: TEST  >F 99  >F 99.987  >F 3.14159267 ;
```

the values will be compiled into the word as floating-point literals. At run-time they will be pushed to the floating-point stack.

8 Displaying Floating-Point Values

F. (F: x --)

Floating-point values can be displayed in a free-format manner using the word **F.**. **F.** works just like its integer counterpart, removing values from the floating-point stack as it displays them.

9 Floating-Point Error Handling

FPERR is a variable that holds the error value for the most recently executed floating-point operation.

?FPERR is a word that, when executed, will display "Floating point error!" and clear the data stack if the most recently executed floating-point operation resulted in an error, *i.e.*, **FPERR** ≠ 0.

10 Floating-Point Stack Manipulation Words

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
FDUP	a — a a	—
FDROP	a —	—
FSWAP	a b — b a	—
FOVER	a b — a b a	—
FPCLEAR	a b ... —	—

10.1 FDUP

Stack Signature: **(F: a -- a a)**

Duplicates the number on top of the floating-point stack.

10.2 FDROP

Stack Signature: **(F: a --)**

Drops the floating-point number on top of the floating-point stack.

10.3 FSWAP

Stack Signature: **`( F: a b -- b a )`**

Swaps the top two floating-point numbers on top of the floating-point stack.

10.4 FOVER

Stack Signature: **`( F: a b -- a b a )`**

Copies the floating-point number immediately beneath the top of the floating-point stack to the top of the floating-point stack.

10.5 FPCLEAR

Stack Signature: **`( F: a b ... -- )`**

Executing **`FPCLEAR`** clears the floating-point stack.

11 Math Words

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
F+	a b — a+b	—
F-	a b — a-b	—
F*	a b — a*b	—
F/	a b — a/b	—
FNEGATE	n — -n	—
FABS	n — |n|	—
FLOOR	x — floor(x)	—
CEIL	x — ceil(x)	—
TRUNC	x — trunc(x)	—
FRAC	x — frac(x)	—

11.1 F+

Stack Signature: **`( F: a b -- a+b )`**

Adds the top two floating-point values on the floating-point stack.

Example:

```
>F 6.594  >F 44.991 F+ F.
```
(displays 51.585)

The above example uses **`>F`** to place 6.594 and 44.991 on the floating-point stack. **`F+`** then adds them, replacing them with the result. **`F.`** then displays the result from the floating-point stack, removing it as it does so.

11.2 F-

Stack Signature: **`( F: a b -- a-b )`**

Subtracts the top two floating-point values on the floating-point stack.

Example:

```
>F 6.594  >F 44.991 F- F.
```
(displays -38.397)

The above example uses **`>F`** to place 6.594 and 44.991 on the floating-point stack. **`F-`** then subtracts the top value from the value immediately beneath it, replacing the values with the result. **`F.`** then displays the result from the floating-point stack, removing it as it does so.

11.3 F*

Stack Signature: **(F: a b -- a*b)**

Multiplies the top two floating-point values on the floating-point stack, replacing them with their product.

Example:

```
>F 6.594  >F 44.991 F* F.      (displays 296.670654)
```

The above example uses **>F** to place 6.594 and 44.991 on the floating-point stack. **F*** then multiplies the top two values, replacing the values with the product. **F.** then displays the result from the floating-point stack, removing it as it does so.

11.4 F/

Stack Signature: **(F: a b -- a/b)**

Divides the floating-point value *a* by the floating-point value *b*, replacing them with the floating-point result.

Example:

```
>F 6.594  >F 44.991 F/ F.      (displays .1465626459)
```

The above example uses **>F** to place 6.594 and 44.991 on the floating-point stack. **F/** then divides the top-most floating-point value into the floating-point value immediately beneath it, replacing them with the result. **F.** then displays the result from the floating-point stack, removing it as it does so.

11.5 FNEGATE

Stack Signature: **(F: a -- -a)**

FNEGATE negates the floating-point number on the floating-point stack, making it negative if it is currently positive, or positive if it is currently negative.

Examples:

```
>F  123456.789 FNEGATE F.     (displays -123456.789)
>F -123456.789 FNEGATE F.     (displays 123456.789)FABS
```

11.6 FABS

Stack Signature: **(F: a -- |a|)**

FABS converts the floating-point number on the floating-point stack to its absolute value.

Examples:

```
>F  123456.789 FABS F.        (displays 123456.789)
>F -123456.789 FABS F.        (displays 123456.789)
```

11.7 FLOOR

Stack Signature: **`( F: x -- floor(x) )`**

`FLOOR` replaces *x* on the floating-point stack with its integer floor, *i.e.*, the greatest integer not greater than *x*.

11.8 CEIL

Stack Signature: **`( F: x -- ceil(x) )`**

`CEIL` replaces *x* on the floating-point stack with its integer ceiling, *i.e.*, the least integer not less than *x*.

11.9 TRUNC

Stack Signature: **`( F: x -- trunc(x) )`**

`TRUNC` replaces *x* on the floating-point stack with the integer part of *x* by truncating *x* at the decimal point.

11.10 FRAC

Stack Signature: **`( F: x -- frac(x) )`**

`FRAC` replaces *x* on the floating-point stack with the fractional part of *x*.

12 Comparison Words

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
F=	x y —	— true|false[1]
F0=	x —	— true|false[1]
F<	x y —	— true|false[1]
F>	x y —	— true|false[1]
F0<	x —	— true|false[1]

*Note: In TurboForth, FALSE is defined as 0. TRUE is defined as any non-zero value. The words defined here represent TRUE as -1.

12.1 F=

Stack Signature: **`( -- true|false ) ( F: x y -- )`**

Returns TRUE on the *data stack* if the floating-point values *x* and *y* are equal, otherwise returns FALSE on the data stack. The floating-point values *x* and *y* are consumed.

Examples:

```
>F 123456.7  >F 123456.7 F= .          (displays -1 (true))
>F 123456.7  >F 123456.8 F= .          (displays 0 (false))
```

12.2 F0=

Stack Signature: **`( -- true|false ) ( F: x -- )`**

Returns TRUE on the *data stack* if the floating-point value *x* is equal to 0.0, otherwise returns FALSE on the data stack. The floating-point value *x* is consumed.

Examples:

```
>F 123456.7  F0= .                     (displays 0 (false))
>F 0  F0= .                        (displays -1 (true))
```

12.3 F<

Stack Signature: **`( -- true|false ) ( F: x y -- )`**

Returns TRUE on the *data stack* if the floating-point value *x* is less than the floating-point value *y*, otherwise returns FALSE on the data stack. The floating-point values *x* and *y* are consumed.

Examples:

```
>F 123456.7 >F 100.1 F< .     (displays 0 (false))
```

1 In TurboForth, FALSE is defined as 0. TRUE is defined as any non-zero value. The words defined here represent TRUE as -1.

```
>F 100.1 >F -99.9 F< .        (displays 0 (false))
>F 100.1 >F 101.1 F< .        (displays -1 (true))
```

12.4 F>

Stack Signature: **(-- true|false) (F: x y --)**

Returns TRUE on the *data stack* if the floating-point value *x* is greater than the floating-point value *y*, otherwise returns FALSE on the data stack. The floating-point values *x* and *y* are consumed.

Examples:

```
>F 123456.7 >F 100.1 F< .     (displays -1 (true))
>F 100.1 >F -99.9 F< .        (displays -1 (true))
>F 100.1 >F 101.1 F< .        (displays 0 (false))
```

12.5 F0<

Stack Signature: **(-- true|false) (F: x --)**

Returns TRUE if the floating-point value *x* is negative.

Examples:

```
>F -123456.0 F0< .                    (displays -1 (true))
>F  123456.0 F0< .                    (displays 0 (false))
```

13 Floating-Point Literal Handling

The following words are discussed in this section:

Word	Floating-point Stack Effect	
	Interpretation	Compilation
>F	— n	—
FLITERAL	n —	—
FLIT	—	— n

13.1 >F

Stack Signature: see section 13.

>F is a state-smart (immediate) word; its behaviour depends upon the state of the compiler:

Compilation State	Interpretation State
The floating point value following **>F** is encoded into the colon-definition as a **FLIT** (floating-point literal) such that at run-time, when the **FLIT** is encountered, the floating-point value is pushed to the floating-point stack.	The floating-point value immediately following **>F** is immediately pushed to the floating-point stack.

The compiler is in compilation state when during the compilation of a colon-definition, or if **]** is executed at the command-line or encountered within a colon-definition.

The compiler is in interpretation-state when typing at the command-line (outside of a colon-definition) or when **[** has been encountered within a colon-definition.

13.2 FLITERAL

Stack Signature: see section 13.

FLITERAL is an immediate word. When encountered during compilation it compiles an **FLIT** instruction to the current compilation address (as pointed to by **HERE**) then the top value from the floating-point stack is moved into the colon-definition immediately after the **FLIT** instruction.

When the word is subsequently executed, the floating-point value will be pushed to the floating-point stack.

Example:

```
: TEST [ >F 3.14159267  >F 2.56  F* FLITERAL ] ;
```

During the compilation of **TEST**, 3.14159267*2.56 is evaluated, and the product (8.042477235) is

directly encoded into the colon-definition as a floating-point literal using **FLIT**. When **TEST** is executed, 8.042477235 shall be pushed to the floating-point stack.

13.3 FLIT

Stack Signature: **(F: -- x)**

FLIT is compiled into definitions by **>F** and **FLITERAL**. When **FLIT** is encountered in a definition, the following 8 bytes (representing a floating-point number in radix-100 format) are pushed to the floating-point stack.

For example, the following code:

```
: TEST >F 3.141 ;
```

Would result in the code-segment **FLIT xx xx xx xx xx xx xx xx** being compiled into the definition (where **xx** represents a radix-100 byte).

14 Floating-Point Variables

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
FVARIABLE	a —	—
F!	value —	address —
F@	— a	address —

14.1 FVARIABLE

Stack Signature: **(F: a --)**

FVARIABLE is the floating point analogue of **VARIABLE**, and works in exactly the same way, allowing one to declare a floating-point type variable of a given name. Like **VARIABLE**, it should not be used inside a colon-definition.

Example:

```
FVARIABLE PI
```

Will declare a floating point variable called **PI** such that whenever **PI** is named subsequently the address of the beginning of the area allocated to the storage of **PI** shall be pushed to the data stack.

Example:

```
PI
```

Causes the address of the beginning of the storage area allocated for **PI** to be pushed to the data stack.

14.2 F!

Stack Signature: **(address --) (F: a --)**

F! is analogous to **!** in standard Forth, and allows a floating-point value to be stored into a floating-point variable previously declared with **FVARIABLE**.

Example:

```
FVARIABLE PI
>F 3.141 PI F!
```

The above will store the floating-point value 3.141 into the floating-point variable **PI**. As described in section 14.1, **PI** causes its associated storage address to be pushed to the data stack, which is used by **F!** in order to store the floating-point value at the correct location.

14.3 F@

Stack Signature: **(address --) (F: -- a)**

F@ is used to retrieve a floating-point value that was previously stored in a floating-point variable with **F!**.

Example:

```
FVARIABLE PI             ( declare PI floating-point variable)
>F 3.141 PI F!           ( store 3.141 in PI)
PI F@                ( retrieve the value stored in PI)
```

Will cause the value 3.141 to be pushed to the floating-point stack.

15 Floating-Point Constants

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
FCONSTANT	value —	—

15.1 FCONSTANT

Stack Signature: **`( F: a -- )`**

`FCONSTANT` declares a floating-point constant, such that when the constant is subsequently named the associated floating-point value is pushed to the floating-point stack.

Example:

```
>F 3.141 FCONSTANT PI
```

The code above pushes 3.141 to the stack and then declares a floating-point constant called **`PI`**, thus associating **`PI`** to the value 3.141. Whenever **`PI`** is then referenced in code or at the command line, 3.141 shall be pushed to the floating-point stack.

`FCONSTANT` should not be used within a colon-definition.

16 Floating-Point Values

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
FVALUE	value —	—
FTO	value —	—
+FTO	value —	—

16.1 FVALUE

Stack Signature: **(F: a --)**

FVALUEs are analogous to **VALUE**s in standard Forth, operating with floating-point numbers rather than integers. **FVALUE** declares a floating-point value, such that when the value is subsequently named, the associated floating-point value is pushed to the floating-point stack.

Unlike **FCONSTANT**s, **FVALUE**s can be be modified with **FTO** and **+FTO**.

Example:

```
>F 7.669 FVALUE CorrectionCoef
```

Here, a floating-point value called **CorrectionCoef** is declared and the value 7.669 is associated with it. Whenever **CorrectionCoef** is referenced either in code at run-time (*i.e.*, in a colon-definition) or at the command line, the value 7.669 shall be pushed to the floating-point stack.

16.2 FTO

Stack Signature: **(F: a --)**

FTO updates the floating-point value in a previously declared **FVALUE**.

Example:

```
>F 7.669 FVALUE CorrectionCoef
: CheckCoef
Temperature 1280 > IF
      >F 9.998 FTO CorrectionCoef
ELSE
      >F 7.669 FTO CorrectionCoef
THEN ;
```

Here, a (presumably non-linear) temperature probe is read, and if the raw value from the probe's ADC exceeds 1280, then a new correction coefficient (9.998) is loaded, otherwise 7.669 is used.

16.3 +FTO

Stack Signature: **(F: a --)**

+FTO adds the value on the floating-point stack to the nominated **FVALUE**.

Example:

```
>F 99.99 FVALUE TEST
>F 100.98 +FTO TEST
```

Causes 100.98 to be added to the current value in **TEST**. That is, after execution, the new value of **TEST** is 200.97.

17 Displaying Floating-Point Numbers

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
F.	a —	—
FF.	a —	left right —
FF+.	a —	left right —
FFE.	a —	left right —
FFE+.	a —	left right —
FFX.	a —	left right —
FFX+.	a —	left right —
F$.	a —	—
.FS	—	—
.F$	—	—

17.1 F.

Stack Signature: **`( F: a -- )`**

`F.` displays the floating-point number *a* on the top of the floating-point stack in a free format determined by the size and type of the number. A 0 will display as an integer 0. An integer of 10 digits or less will display with no decimal point. Non-integers will display in a floating decimal format if at least 6 significant digits can be displayed. Otherwise, the number is displayed in scientific notation with up to 6 significant digits and "x10^{xx}" represented by "Esxx", where 'xx' are the digits of the power of 10 and 's' its sign. If 'xx' is actually 3 digits, they will be represented by '**' even though the number may be a proper floating-point number on the stack.

Examples:

`>F 123.456 F.` (displays 123.456)
`>F 0 F.` (displays 0)
`>F 123 F.` (displays 123)
`>F -123.45678912345 F.` (displays -123.4567891)
`>F 12345678901234 F.` (displays 1.23457E+13)
`>F 1.23456E122 F.` (displays 1.23456E+**)
`>F 0.00000123456 F.` (displays 1.23456E-06)

17.2 FF.

Stack Signature: **`( left right -- ) ( F: a -- )`**

`FF.` displays the floating-point number *a* on top of the floating-point stack in a fixed format with *left* digits to the left of the decimal point, including the sign, and *right* digits to the right of the

decimal point, including the decimal point, where *left* + *right* ≤ 14. If the formatted number cannot be displayed with the format provided, a string of *left* + *right* asterisks (*) will be displayed. Nothing is printed and **FPERR** is set if *left* + *right* > 14.

Examples:

```
>F -123.45678912345 4 10 FF.  (displays -123.456789124)
>F 3.456E26 2 12 FF.  (displays **********)
>F -0.98765 3 7 FF.  (displays  -.987650)
```

17.3 FF+.

Stack Signature: **(left right --) (F: a --)**

FF+. is identical to **FF.** with the addition of the display of an explicit '+' instead of a blank for positive numbers.

Example:

```
>F 123.4567 4 7 FF+.  (displays +123.456700)
```

17.4 FFE.

Stack Signature: **(left right --) (F: a --)**

FFE. formats the floating-point number *a* on the floating-point stack in E-notation, but is otherwise identical to **FF.** format. The number display is 4 characters wider to make room for the E-notation with the same format as for §17.1 F., and is not included in the *left* + *right* calculation.

Examples:

```
>F 123.4567 2 7 FFE.  (displays  1.234567E+02)
>F -1.234567E-12 3 7 FFE.  (displays -12.345670E-13)
>F 1.234567E109 2 7 FFE.  (displays  1.234567E+**)
```

17.5 FFE+.

Stack Signature: **(left right --) (F: a --)**

FFE+. is identical to **FFE.** with the addition of the display of an explicit '+' instead of a blank for positive numbers.

Example:

```
>F 123.4567 2 7 FFE+.  (displays +1.234567E+02)
```

17.6 FFX.

Stack Signature: **(left right --) (F: a --)**

FFX. is identical to **FFE.** with the exponent extended to a third place to accommodate the maximum and minimum possible exponents (-128 – 127).

Example:

```
>F 9.87654321E-109 2 11 FFX.  (displays  9.8765432100E-109)
```

17.7 FFX+.

Stack Signature: **(left right --) (F: a --)**

FFX+. is identical to **FFX.** with the addition of an explicit '+' instead of a blank for positive numbers.

Example:

>F 3.810236E22 2 7 FFX+. (displays +3.810236E+022)

17.8 F$.

Stack Signature: **(F: a --)**

F$. displays in hexadecimal format the floating-point number on top of the floating-point stack exactly as it is stored, *i.e.*, 4 cells (8 bytes).

Example:

>F 3.810236E22 F$. (displays 4B03 5102 2400 0000)

17.9 .FS

Stack Signature: **(--)**

.FS displays all the numbers on the floating-point stack, without removing them, in the same free format as **F.** (see §17.1). It is the floating-point equivalent of **.S** . For example, with the same 6 numbers as in §17.1 on the floating-point stack, the following would be the result:

```
.FS  123.456  0  123 -123.4567891  1.23457E+13  1.23456E+**
1.23456E-06 <--TOP
ok:0
```

17.10 .F$

Stack Signature: **(--)**

.F$ displays all the numbers on the floating-point stack in hexadecimal format, one floating-point number per line, exactly as they are stored on the floating-point stack, without removing them. This results in essentially a dump of the floating-point stack as 8 hexadecimal bytes (4 cells) per line. The same floating-point stack contents as in §17.9 would result in:

```
.F$
4101 172D 3C00 0000
0000 0000 6000 0460
4101 1700 0000 0000
BEFF 172D 4359 0C23
460C 2238 4E5A 0C22
7D01 172D 3C00 0000
3D01 172D 3C00 0000 <--TOP
ok:0
```

18 Floating-Point Number Conversion

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
S>FP	— a	a —
FP>S	a —	— a

18.1 S>FP

Stack Signature: **`( a -- ) ( F: -- a )`**

`S>FP` takes an integer cell (a “single”) from the data stack, converts to floating-point and leaves it on the floating-point stack.

Example:

```
100 S>FP
```

Results in 100.00 being placed on the floating-point stack.

18.2 FP>S

Stack Signature: **`( -- a ) ( F: a -- )`**

`FP>S` converts the floating-point number on top of the floating-point stack to an integer on the data stack.

Example:

```
>F 102.567 FP>S
```

Results in 103 being placed on the data stack.

19 Transcendental Constants and Conversion Functions

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
PI	— π	—
EULER_E	— *e*	—
RAD/DEG	— a	—
DEG/RAD	— a	—
>RAD	deg — rad	—
>DEG	rad — deg	—

19.1 PI

Stack Signature: **`( F: -- π )`**

`PI` is a floating-point constant that will deposit on the floating-point stack the value of π to 13 significant digits (3.141592653590).

19.2 EULER_E

Stack Signature: **`( F: -- e )`**

`EULER_E` is a floating-point constant that will deposit on the floating-point stack the value of *e* to 13 significant digits (2.718281828459).

19.3 RAD/DEG

Stack Signature: **`( F: -- a )`**

`RAD/DEG` is a floating-point constant that will deposit on the floating-point stack the value of π/180 radians/degree to 13 significant digits (0.01745329251994).

19.4 DEG/RAD

Stack Signature: **`( F: -- a )`**

`DEG/RAD` is a floating-point constant that will deposit on the floating-point stack the value of 180/π degrees/radian to 14 significant digits (57.295779513082).

19.5 >RAD

Stack Signature: **`( F: deg -- rad )`**

`>RAD` converts the floating-point number *deg* degrees on the floating-point stack to *rad* radians on the floating-point stack.

19.6 >DEG

Stack Signature: **`( F: rad -- deg )`**

`>DEG` converts the floating-point number *rad* radians on the floating-point stack to *deg* degrees on the floating-point stack.

20 Transcendental Functions

The following words are discussed in this section:

Word	Floating-point Stack Effect	Data Stack Effect
EXP	x — e^x	—
LOG	x — $\log_e x$	—
SQRT	x — $\sqrt{x}$	—
COS	x — cos(x)	—
SIN	x — sin(x)	—
TAN	x — tan(x)	—
ATN	x — atn(x)	—
POW	base exp — $base^{exp}$	—
LOG10	x — $\log_{10} x$	—
EXP10	x — 10^x	—

20.1 EXP

Stack Signature: **(F: x -- e^x)**

EXP replaces *x* on the floating-point stack with e^x.

20.2 LOG

Stack Signature: **(F: x -- $\log_e x$)**

LOG replaces *x* on the floating-point stack with its natural logarithm $\log_e x$.

20.3 SQRT

Stack Signature: **(F: x -- $\sqrt{x}$)**

SQRT replaces *x* on the floating-point stack with its square root $\sqrt{x}$.

20.4 COS

Stack Signature: **(F: x -- cos(x))**

COS replaces *x* on the floating-point stack with its cosine cos(*x*).

20.5 SIN

Stack Signature: **(F: x -- sin(x))**

SIN replaces *x* on the floating-point stack with its sine sin(*x*).

20.6 TAN

Stack Signature: **(F: x -- tan(x))**

TAN replaces *x* on the floating-point stack with its tangent tan(*x*).

20.7 ATN

Stack Signature: **(F: x -- atn(x))**

ATN replaces *x* on the floating-point stack with its arctangent atn(*x*) or $\tan^{-1}(x)$.

20.8 POW

Stack Signature: **(F: base exp -- base^{exp})**

POW pops *base* and *exp* off of the floating-point stack and replaces them with $base^{exp}$.

20.9 LOG10

Stack Signature: **(F: x -- log$_{10}$x)**

LOG10 replaces *x* on the floating-point stack with its common logarithm $\log_{10}x$.

20.10 EXP10

Stack Signature: **(F: x -- 10^x)**

EXP10 replaces *x* on the floating-point stack with 10^x.

21 Source Code

The following sections list the source code that comprises the floating-point library. The source code is presented here for general interest and understanding. It also provides a source base if the reader wishes to make modifications or add extra functionality.

Note that it is not really practical to store the assembler source code in block format because it would require that the assembler be loaded and would take more room. It is possible to use the TurboForth assembler to write assembly code in a horizontal format, similar to high level Forth, so that it will load faster, but it makes the assembler code very difficult to read.

TurboForth contains a word in its standard dictionary called **CODE:** which permits the definition of an assembly language word using numeric values only. This uses vastly less space in a block than its assembly source equivalent in horizontal format. Additionally, it gains even more in load speed because TurboForth does not need to translate assembly language mnemonics into machine code op-codes, as with the assembler source code. **CODE:** versions of all assembly language coded words are presented here. The **CODE:** versions of the assembly language words were produced with a small utility called **ASM>CODE** . See §23 for more information.

Finally, a version suitable for inclusion on disk blocks is presented in §22 .

21.1 Start-Up Script

Block 54 of the BLOCKS file below starts by defining the word **load-fpl** to load the memory image of the FPL from blocks 66 – 71 into the beginning of CPU RAM's low-memory at 2000h, which it then proceeds to execute. The definition of **load-fpl** is removed, **ffailm** is adjusted to point just beyond the end of the FPL machine code and **HERE** is adjusted to point to the same location:

```
DECIMAL CR .( Loading floating-point library...)
.( Machine code library...)
: load-fpl ( block--) 6 0 do i over + block i 1024 * $2000 +
  1024 vmbr loop drop 54 block tib ! ;
66 load-fpl  forget load-fpl  $366C ffailm !  ffailm @ h !
```

21.2 Necessary Values, Constants, Variables and Arrays

The script continues with the rest of block 54 and continues through block 65.

21.2.1 FPSTAT

```
VARIABLE FPSTAT   \ FPLTF return status
```

21.2.2 FPERR

```
VARIABLE FPERR          \ FP error variable
```

21.2.3 _fpssz

```
10 CONSTANT _fpssz      \ max number of entries on FP stack
```

21.2.4 _fpsc

```
0 VALUE    _fpsc        \ number of items on fp stack
```

21.2.5 _fpstack

```
CREATE _fpstack _fpssz 8 * ALLOT   \ floating point stack
```

21.2.6 _fptemp

```
CREATE _fptemp 8 CHARS ALLOT       \ temporary FP storage
```

21.2.7 _fpstr

```
CREATE _fpstr 24 CHARS ALLOT       \ FP string buffer
```

21.2.8 _fpsaddr

```
VARIABLE _fpsaddr                  \ address of top of FP stack
_fpstack _fpsaddr !                \ initialize
```

21.3 Floating Point Error Handling

21.3.1 ?FPERR

```
: ?FPERR ( -- )  \ report floating point error
      FPERR @ ABORT"  Floating point error!" ;
```

21.4 Internal Routines

21.4.1 goFPL

This routine links to the Floating Point Library for TurboForth.

```
\ ---expects _fpstr already set up
ASM: goFPL ( FPaddr1 [FPaddr2] [CNSopt1 CNSopt2 CNSopt3] FPL_fxn -- )
    *SP+ R0 MOV,         \ get function # to R0
    R0 18 CI,            \ CNS?
    EQ IF,               \ yes; get 3 options
        *SP+ R9 MOV,     \ get CNSopt3
        *SP+ R8 MOV,     \ get CNSopt2
        *SP+ R7 MOV,     \ get CNSopt1
    ENDIF,
    *SP+ R1 MOV,         \ point to 1st FPaddr
    R0 14 CI,            \ conversion routines?
    GT IF,               \ yes
        R2 _fpstr LI,    \ point R2 to FP string buffer
    ELSE,
        R0 R0 MOV,       \ COMPARE routine?
        EQ IF,           \ yes
            *SP+ R2 MOV,    \ point to 2nd FPaddr
        ELSE,
            R0 5 CI,     \ 2-parameter function?
            LE IF,       \ yes
```

```
                *SP+ R2 MOV,    \ point to 2nd FPaddr
            ENDIF,
        ENDIF,
    ENDIF,
    $2000 @@ BLWP,     \ link to FPL
    R7 STST,            \ get FP status
    R7 FPSTAT @@ MOV,
    R0 FPERR @@ MOV,    \ get FP error
    \ restore scratchpad memory
    $6000 @@ CLR,       \ select bank 1
    $A010 @@ R0 MOV,    \ scratchpad restore code vector
    R0 ** BL,           \ restore scratchpad
    $6002 @@ CLR,       \ select bank 0
;ASM
```

Code Version

```
\ ...asm>code code patched with named variables and constants
HEX
CODE: goFPL C034 0280 0012 1603 C274 C234 C1F4 C074 0280 000E
1501 1003 0202 _fpstr , 1008 C000 1602 C0B4 1004 0280 0005 1B01
C0B4 0420 2000 02C7 C807 fpstat , C800 fperr , 04E0 6000 C020
A010 0690 04E0 6002 ;CODE
DECIMAL
```

21.4.2 _fpchg

This routine increments, decrements or clears the floating-point stack. It returns an error flag on the stack to indicate no error (0), FP stack overflow (+1) or FP stack underflow (-1). This routine should be followed with **?fpserr** (see next section) wherever it is used in order to properly handle errors.

```
ASM: _fpchg ( chg -- flag ) \ size change is +1, -1 or 0 to clear
                            \ flag: 0=no error; 1=over; -1=under
    *SP R0 MOV, \ get change
    EQ IF,   \ is 0
        _fpsc @@ CLR,   \ zero _fpsc
        R0 _fpstack LI, \ get FP stack bottom
        R0 _fpsaddr @@ MOV,  \ set _fpsaddr to FP stack bottom
    ELSE,   \ not 0
        GT IF,  \ increment stack
            _fpsc @@ R1 MOV,
            R1 _fpssz CI,
            NE IF,  \ < max?
                *SP CLR,    \ clear flag
                R0 1 LI,    \ ensure only 1 more [OVERKILL??]
            ENDIF,
        ELSE,   \ decrement stack
            _fpsc @@ R1 MOV,
            NE IF,  \ <> 0?
                *SP CLR,    \ clear flag
                R0 -1 LI,   \ ensure only 1 less [OVERKILL??]
            ENDIF,
        ENDIF,
        *SP *SP MOV,    \ OK to change stack count?
        EQ IF,  \ OK to change stack count
            R0 _fpsc @@ A,  \ +/-1 to FP item #
```

```
            R0 3 SLA,       \ x8
            R0 _fpsaddr @@ A, \ inc _fpsaddr 8 bytes
        ENDIF,
    ENDIF,
;ASM
```

Code Version

```
\ ...asm>code code patched with named variables and constants
HEX
CODE: _fpchg C014 1607 04E0 _fpsc , 0200 _fpstack , C800
_fpsaddr , 1018 1501 1009 C060 _fpsc , 0281 _fpssz , 1303 04D4
0200 0001 1006 C060 _fpsc , 1303 04D4 0200 FFFF C514 1605 A800
_fpsc , 0A30 A800 _fpsaddr , ;CODE
DECIMAL
```

21.4.3 ?fpserr

```
: ?fpserr ( flag -- ) \ fp stack over/underflow message and abort
    ?DUP IF
        ." Floating-point stack " 0> IF
            ." ov"
        ELSE
            ." und"
        THEN
        ." erflow!" CR ABORT
    THEN
;
```

21.4.4 fpsc++

```
: fpsc++ (  --  ) \ increase fp stack pointer
    \ flag = 0, no error; flag = 1, overflow
        1 _fpchg ?fpserr
;
```

21.4.5 fpsc--

```
: fpsc-- (  --  ) \ decrease fp stack pointer
    \ flag = 0, no error; flag = -1, underflow
    -1 _fpchg ?fpserr
;
```

21.4.6 str>fpstr

```
: str>fpstr
    \ move string in input stream to FP string buffer
    BL WORD DUP _fpstr C! _fpstr 1+ SWAP BLK @ IF
        VMBR ELSE CMOVE
    THEN
    0 _fpstr DUP C@ + 1+ C! \ terminate string with NULL
;
```

21.4.7 doComp

```
: doComp  ( mask -- n ) ( F: a b -- )
    \ Status byte bits for mask:
    \ | 0  |  1 | 2 |   3   |   4   | 5 | 6 | 7 |
    \ | L> | A> | = | CARRY | OFLOW | OP| X | ? |
    \ invoke goFPL with 2 variables on FP stack and return error on stack
    \ and statusByte mask on data stack
    fpsc-- _fpsaddr @ fpsc-- _fpsaddr @ SWAP 0 goFPL
    FPSTAT C@ \ status is in left byte
    AND
;
```

21.4.8 fpstr>fp

```
: fpstr>fp
    \ convert string in FP string buffer to FP number on FP stack
    _fpsaddr @ fpsc++ $0011 goFPL
;
```

21.4.9 fpstr>here

```
: fpstr>here
    \ convert string in FP string buffer to HERE
    \ fp value placed at HERE
    HERE $0011 goFPL
;
```

21.5 Floating-Point Stack Manipulation Words

21.5.1 FDUP

```
: FDUP ( F: n -- n n )
    fpsc++ _fpsaddr @ 16 - _fpsaddr @ 8 - 4 COPYW
;
```

21.5.2 FDROP

```
: FDROP ( F: n -- )
    fpsc--
;
```

21.5.3 FSWAP

```
: FSWAP ( F: a b -- b a )
    _fpsaddr @ 8 - _fptemp 4 COPYW
    _fpsaddr @ 16 - _fpsaddr @ 8 - 4 COPYW
    _fptemp _fpsaddr @ 16 - 4 COPYW
;
```

21.5.4 FOVER

```
: FOVER ( F: a b -- a b a )
    fpsc++ _fpsaddr @ 24 - _fpsaddr @ 8 - 4 COPYW
;
```

21.5.5 FPCLEAR

```
: FPCLEAR ( -- )
    \ clear floating point stack
    0 _fpchg ?fpserr
;
```

21.6 Floating-Point Math Words

21.6.1 ndf0<

```
: ndf0< ( -- true|false ) ( F: n -- n )
    \ non-destructive, floating-point 0<
    _fpsaddr @ 8 - @ 0<
;
```

21.6.2 F+

```
: F+ ( F: a b -- a+b )
    fpsc-- _fpsaddr @ DUP 8 - SWAP $0002 goFPL
;
```

21.6.3 F-

```
: F- ( F: a b -- a-b )
    fpsc-- _fpsaddr @ DUP 8 - SWAP $0001 goFPL
;
```

21.6.4 F*

```
: F* ( F: a b -- a*b )
    fpsc-- _fpsaddr @ DUP 8 - SWAP $0003 goFPL
;
```

21.6.5 F/

```
: F/ ( F: a b -- a\b )
    fpsc-- _fpsaddr @ DUP 8 - SWAP $0004 goFPL
;
```

21.6.6 FNEGATE

```
: FNEGATE ( F: n -- -n )
    _fpsaddr @ 8 - DUP @ NEGATE SWAP !
;
```

21.6.7 FABS

```
: FABS ( F: n -- |n| )
    ndf0< IF
        FNEGATE
    THEN
;
```

21.6.8 FLOOR

```
: FLOOR ( F: x -- floor[x] )
    _fpsaddr @ 8 - $000D goFPL
;
```

21.6.9 CEIL

This definition of **CEIL** must follow the definition of **>F** below.

```
: CEIL ( F: x -- ceil[x] )
    FLOOR >F 1 F+
;
```

21.6.10 TRUNC

This definition of **TRUNC** must follow the definition of **>F** below.

```
: TRUNC ( F: x -- trunc[x] )
    FLOOR  ndf0< IF
        >F 1 F+
    THEN
;
```

21.6.11 FRAC

This definition of **FRAC** must follow the definition of **>F** below and **TRUNC** above.

```
: FRAC  ( F: x -- trunc[x] )
    FDUP TRUNC F-
;
```

21.7 Floating-Point Literal Handling

21.7.1 FLIT

```
: FLIT  ( F: -- n )
    \ copy the radix 100 string immediately after the FLIT
    \ opcode to fp stack
    \ addr on return stack points to payload copy it to FP stack
    R@ _fpsaddr @ 4 COPYW fpsc++
    R> 8 + >R    \ move return address past payload
;
```

21.7.2 FLITERAL

```
: FLITERAL  ( F: n -- )
    COMPILE FLIT  fpsc--  _fpsaddr @ HERE 4 COPYW
    8 ALLOT ( advance HERE past the fp value)
; IMMEDIATE
```

21.7.3 >F

```
: >F  ( F: -- n )
    \ If state=0:
    \   converts a number in the input stream to FP and places
    \   it on the FP stack
    \ If state=1:
    \   compiles FLIT and copies the radix 100 8-byte stream to HERE

    STATE @ IF
        \ we're compiling
        COMPILE FLIT    \ compile reference to FLIT
        str>fpstr       \ copy string in input stream to FP string buffer
        fpstr>here      \ convert string in FP string buffer to FP at HERE
        8 ALLOT         \ move past the radix-100 string
    ELSE
        str>fpstr
        fpstr>fp
    THEN
; IMMEDIATE
```

21.8 Floating-Point Comparison Words

```
\ Status byte bits for mask passed to doComp:
\ | 0  |  1 | 2 |   3   |   4   | 5 | 6 | 7 |
\ | L> | A> | = | CARRY | OFLOW | OP| X | ? |
```

21.8.1 F=

```
: F=  ( -- true|false ) ( F: a b -- )
    %00100000 doComp 0>
;
```

21.8.2 F0=

```
: F0= ( -- true|false ) ( F: a -- )
    fpsc-- _fpsaddr @ @ 0=
;
```

21.8.3 F>

```
: F>  ( -- true|false ) ( F: a b -- )
    %01000000 doComp 0>
;
```

21.8.4 F<

```
: F<  ( -- true|false ) ( F: a b -- )
    %01100000 doComp 0=
;
```

21.8.5 F0<

```
: F0< ( -- true|false ) ( F: a -- )
    fpsc-- _fpsaddr @ @ 0<
;
```

21.9 Floating-Point Variables

21.9.1 FVARIABLE

```
: FVARIABLE
    CREATE 8 CHARS ALLOT
;
```

21.9.2 F!

```
: F! ( address -- ) ( F: n -- )
    fpsc-- _fpsaddr @ SWAP 4 COPYW
;
```

21.9.3 F@

```
: F@ ( address -- ) ( F: -- n )
     _fpsaddr @ 4 COPYW  fpsc++
;
```

21.10 Floating-Point Constants

21.10.1 FCONSTANT

```
: FCONSTANT ( F: n -- )
    CREATE 8 ALLOT
    fpsc-- _fpsaddr @ HERE 8 -  4 COPYW
    DOES> _fpsaddr @ 4 COPYW fpsc++
;
```

21.11 Floating-Point Values

21.11.1 FVALUE

```
: FVALUE ( F: n -- )
    FCONSTANT
;
```

21.11.2 (FTO)

```
: (FTO) ( body -- )
    fpsc-- _fpsaddr @ SWAP 4 COPYW
;
```

21.11.3 FTO

```
: FTO ( F: n -- )
    \ set FP value to n. e.g: >F 1.234 FTO X
    BL WORD FIND DROP >BODY   \ address of r100 data
    STATE @ IF
        [COMPILE] LITERAL  COMPILE (FTO)
    ELSE
        (FTO)
    THEN
; IMMEDIATE
```

21.11.4 (+FTO)

```
: (+FTO) ( body -- )
    DUP                         \ dup body addr
    _fpsaddr @ 4 COPYW fpsc++  \ move r100 value in the body to FP stack
    F+                          \ perform addition
    fpsc-- _fpsaddr @ SWAP 4 COPYW   \ copy result to body
;
```

21.11.5 +FTO

```
: +FTO ( F: n -- )
    \ the FP value n is added to the FP FVALUE
    BL WORD FIND DROP >BODY
    STATE @ IF
        [COMPILE] LITERAL  COMPILE (+FTO)
    ELSE
        (+FTO)
    THEN
; IMMEDIATE
```

21.12 Displaying Floating-Point Numbers (except fixed format)

21.12.1 F.

```
: F. ( F: n -- )
    \ display the top FP number in decimal
    fpsc-- _fpsaddr @   \ pop top of fp stack
    0 0 0               \ free format output (CNSopt1 CNSopt2 CNSopt3)
    $0012 goFPL         \ convert fp number to string at _fpstr
    _fpstr COUNT TYPE   \ display it
;
```

21.12.2 .FS

```
: .FS ( -- )
    \ non-destructively display contents of FP stack
    _fpsc @ 0> IF
        _fpstack
        _fpsc @ 0 DO
            DUP
            0 0 0           \ free format output (CNSopt1 CNSopt2 CNSopt3)
            $0012 goFPL     \ convert FP number to string at _fpstr
            _fpstr COUNT TYPE SPACE   \ display FP number
            8 +             \ increment to next FP number on FP Stack
        LOOP DROP ." <--TOP" CR
    ELSE
        ." Empty "
    THEN
;
```

21.12.3 .F$

```
: .F$ ( -- )
    \ display fp stack in hex, 8 bytes/line
    _fpsc @ 0> IF
        TRUE ZEROS !
        _fpstack DUP _fpsc @ 8 * + SWAP DO
            CR I 8 + I DO
                I @ $.
            2 +LOOP
        8 +LOOP ." <--TOP" CR FALSE ZEROS !
    ELSE
        ." Empty "
    THEN
;
```

21.12.4 F$.

```
: F$. ( F: n -- )
    \ display FP number on top of stack in R100
    fpsc-- TRUE ZEROS ! _fpsaddr @ DUP 8 + SWAP DO
        I @ $.
    2 +LOOP
    FALSE ZEROS !
;
```

21.13 Floating-Point Number Conversion

21.13.1 S>FP

```
: S>FP ( n -- ) ( F: -- n )
    \ convert single to floating point number
    _fpsaddr @ DUP -ROT fpsc++ ! $000F goFPL
;
```

21.13.2 FP>S

```
: FP>S    ( -- n ) ( F: n -- )
    fpsc-- _fpsaddr @ DUP $000E goFPL @
;
```

21.14 Transcendental Constants and Conversion Functions

This section and the next are out of order with respect to their appearance in the dictionary. See Chapter 22 Source Code – Blocks Version for the actual word order.

21.14.1 PI

This constant is the last word in low memory, leaving only 30 bytes free.

```
>F 3.141592653590 FCONSTANT PI
```

21.14.2 RAD/DEG

This constant and all succeeding words in the floating-point library are stored in high memory.

```
>F 0.01745329251994 FCONSTANT RAD/DEG
```

21.14.3 DEG/RAD

```
>F 57.295779513082 FCONSTANT DEG/RAD
```

21.14.4 EULER_E

```
>F 2.718281828459 FCONSTANT EULER_E
```

21.14.5 >RAD

```
: >RAD ( F: deg -- rad )
    RAD/DEG F*
;
```

21.14.6 >DEG

```
: >DEG ( F: rad -- deg )
    DEG/RAD F*
;
```

21.15 Transcendental Functions

21.15.1 EXP

```
: EXP ( F: x -- exp[x] )
    _fpsaddr @ 8 - $0006 goFPL
;
```

21.15.2 LOG

```
: LOG ( F: x -- log[x] )
    _fpsaddr @ 8 - $0007 goFPL
;
```

21.15.3 SQRT

```
: SQRT ( F: x -- sqrt[x] )
    _fpsaddr @ 8 - $0008 goFPL
;
```

21.15.4 COS

```
: COS ( F: x -- cos[x] )
    _fpsaddr @ 8 - $0009 goFPL
;
```

21.15.5 SIN

```
: SIN ( F: x -- sin[x] )
    _fpsaddr @ 8 - $000A goFPL
;
```

21.15.6 TAN

```
: TAN ( F: x -- tan[x] )
    _fpsaddr @ 8 - $000B goFPL
;
```

21.15.7 ATN

```
: ATN ( F: x -- atn[x] )
    _fpsaddr @ 8 - $000C goFPL
;
```

21.15.8 POW

```
: POW ( F: base exp --- base^exp )
    fpsc-- _fpsaddr @ DUP 8 - SWAP $0005 goFPL
;
```

21.15.9 LOG10

```
: LOG10 ( F: x -- log10[x] )
    LOG >F 0.43429448190325 F*
;
```

21.15.10 EXP10

```
: EXP10 ( F: x -- 10^x )
    >F 10 FSWAP POW
;
```

21.16 Displaying Floating-Point Numbers in Fixed Format

21.16.1 fixFmt.

```
\ fixed format display routines:
\   *intLen places to left of decimal point, including sign
\   *fracLen places to right of decimal point, including decimal
\    point
```

```
\   *mask includes bits (sign, E-notation options) to be ORed
\   with CNSopt1 = 1
.(   Fixed format display routines...)
: fixFmt. ( intLen fracLen mask -- ) ( F: n -- )
    \ display the top fp number in fixed decimal format
    fpsc-- _fpsaddr @   \ pop top of fp stack and get address
    SWAP 1 OR           \ fixed format output + sign & E-notation options
    3 ROLL 3 ROLL       \ put in order:FPaddr1 CNSopt1 CNSopt2 CNSopt3
    $0012 goFPL         \ convert fp number to string at _fpstr
    _fpstr COUNT TYPE   \ display it
;
```

21.16.2 FF.

```
\ high-level fixed format output words that depend on fixFmt.

: FF.  ( intLen fracLen -- ) ( F: n -- )
    \ display fixed format FP value n
    0 fixFmt.
;
```

21.16.3 FF+.

```
: FF+.  ( intLen fracLen -- ) ( F: n -- )
    \ display format FP value n with '+'
    %110 fixFmt.
;
```

21.16.4 FFE.

```
: FFE.  ( intLen fracLen -- ) ( F: n -- )
    \ display E-notation fixed format FP value n
    %1000 fixFmt.
;
```

21.16.5 FFE+.

```
: FFE+.  ( intLen fracLen -- ) ( F: n -- )
    \ display E-notation fixed format FP value n with '+'
    %1110 fixFmt.
;
```

21.16.6 FFX.

```
: FFX.   ( intLen fracLen -- ) ( F: n -- )
    \ display extended E-notation fixed format FP value n
    %11000 fixFmt.
;
```

21.16.7 FFX+.

```
: FFX+.  ( intLen fracLen -- ) ( F: n -- )
    \ display extended E-notation fixed format FP value n with '+'
    %11110 fixFmt.
;
```

22 Source Code – Blocks Version

Block 54 begins with the definition of **load-fpl** , which is then invoked to load the FPL memory image stored in blocks 66 – 71 into low memory starting at 2000h. Next, the definition of **load-fpl** is removed from the dictionary, the value of **ffailm** is adjusted to the end of the memory image, **HERE** is updated to that value and the remainder of the blocks is loaded with the high-level floating-point Forth definitions.

The header of each block below begins with "**#nn**", where "**nn**" is the block number and the remaining marks indicate columns 0 – 63. Down the left side are the line numbers as "**nn|**". Neither the headers nor the line numbers are part of the code.

22.1 DSK1.BLOCKS (Blocks 54 – 65)

```
#54-----+----1----+----2----+----3----+----4----+----5----+----6---
00|DECIMAL CR .( Loading floating-point library...)
01|.( Machine code library...)
02|: load-fpl ( block--) 6 0 do i over + block i 1024 * $2000 +
03|  1024 vmbr loop drop 54 block tib ! ;
04|66 load-fpl  forget load-fpl  $366C ffailm !  ffailm @ h !
05|.(   General support code...)
06|VARIABLE FPSTAT  \ FPLTF return status
07|VARIABLE FPERR   \ FP error
08|: ?FPERR ( -- )  FPERR @ ABORT"  Floating-point error!" ;
09|10 CONSTANT _fpssz   \ max number of entries on FP stack
10|VARIABLE _fpsc    \ number of items on fp stack
11|CREATE _fpstack _fpssz 8 * ALLOT  \ floating point stack
12|CREATE _fptemp 8 ALLOT
13|CREATE _fpstr 24 ALLOT \ fp string buffer
14|VARIABLE _fpsaddr \ address of top of FP stack
15|_fpstack _fpsaddr !    HEX -->

#55-----+----1----+----2----+----3----+----4----+----5----+----6---
00|CODE: goFPL C034 0280 0012 1603 C274 C234 C1F4 C074 0280 000E
01|1501 1003 0202 _fpstr , 1008 C000 1602 C0B4 1004 0280 0005 1B01
02|C0B4 0420 2000 02C7 C807 fpstat , C800 fperr , 04E0 6000 C020
03|A010 0690 04E0 6002 ;CODE
04|CODE: _fpchg C014 1607 04E0 _fpsc , 0200 _fpstack , C800
05|_fpsaddr , 1018 1501 1009 C060 _fpsc , 0281 _fpssz , 1303 04D4
06|0200 0001 1006 C060 _fpsc , 1303 04D4 0200 FFFF C514 1605 A800
07|_fpsc , 0A30 A800 _fpsaddr , ;CODE  decimal
08|: ?fpserr ( flag -- ) \ fp stack over/underflow msg and abort
09|  ?DUP IF ." Floating-point stack " 0> IF ." ov" ELSE ." und"
10|  THEN ." erflow!" CR ABORT THEN ;
11|: fpsc++ (  --  ) \ increase fp stack pointer
12|  ( flag = 0, no error; flag = 1, overflow) 1 _fpchg ?fpserr ;
13|: fpsc-- (  --  ) \ decrease fp stack pointer
14|  ( flag=0, no error; flag = -1, underflow) -1 _fpchg ?fpserr ;
15|-->
```

```
#56-----+----1----+----2----+----3----+----4----+----5----+----6---
00|: str>fpstr \ move string in input stream to FP string buffer
01|  BL WORD DUP _fpstr C! _fpstr 1+ SWAP BLK @ IF VMBR ELSE
02|  CMOVE THEN 0 _fpstr DUP C@ + 1+ C! ; \ terminate with NULL
03|: doComp  ( ds:mask -- n fs:a b -- )
04|  fpsc-- _fpsaddr @ fpsc-- _fpsaddr @ SWAP 0 goFPL
05|  FPSTAT C@ ( status is in left byte) AND ;
06|: fpstr>fp \ convert string in FP string buffer to fp number
07|  ( fp value placed in _fac) _fpsaddr @ fpsc++ $0011 goFPL ;
08|: fpstr>here \ convert string in FP string buffer to HERE
09|  ( fp value placed in _fac) HERE $0011 goFPL ;
10|.(   Floating-point stack manipulation...)
11|: FDUP ( ds: --  fs:n -- n n )
12|  fpsc++ _fpsaddr @ 16 - _fpsaddr @ 8 - 4 COPYW ;
13|: FDROP ( ds: --  fs:n -- ) fpsc-- ;
14|: FOVER ( ds: --  fs:a b -- a b a )
15|  fpsc++ _fpsaddr @ 24 - _fpsaddr @ 8 - 4 COPYW ;  -->

#57-----+----1----+----2----+----3----+----4----+----5----+----6---
00|: FSWAP ( ds: --  fs:a b -- b a ) _fpsaddr @ 8 - _fptemp 4 COPYW
01|  _fpsaddr @ 16 - _fpsaddr @ 8 - 4 COPYW _fptemp _fpsaddr @ 16 -
02| 4 COPYW ;
03|: FPCLEAR ( --) ( clear fp stack) 0 _fpchg ?fpserr ;
04|
05|.(   Floating-point math...)
06|\ non-destructive, floating-point 0<
07|: ndf0< ( -- true|false ) ( F: n -- n ) _fpsaddr @ 8 - @ 0< ;
08|: F+ ( ds: --  fs:a b -- a+b )
09|  fpsc-- _fpsaddr @ DUP 8 - SWAP $0002 goFPL ;
10|: F- ( ds: --  fs:a b -- a-b )
11|  fpsc-- _fpsaddr @ DUP 8 - SWAP $0001 goFPL ;
12|: F* ( ds: --  fs:a b -- a*b )
13|  fpsc-- _fpsaddr @ DUP 8 - SWAP $0003 goFPL ;
14|: F/ ( ds: --  fs:a b -- a\b )
15|  fpsc-- _fpsaddr @ DUP 8 - SWAP $0004 goFPL ;  -->

#58-----+----1----+----2----+----3----+----4----+----5----+----6---
00|: FNEGATE ( ds: --  fs:n -- -n )
01|  _fpsaddr @ 8 - DUP @ NEGATE SWAP ! ;
02|: FABS ( F: n -- |n| ) ndf0< IF FNEGATE THEN ;
03|: FLOOR ( f: x -- floor[x] ) _fpsaddr @ 8 - $000D goFPL ;
04|
05|.(   Floating point literal handling...)
06|: FLIT ( ds: -- fs: -- n)
07|  R@ _fpsaddr @ 4 COPYW fpsc++
08|  R> 8 + >R ( move return address past payload) ;
09|: FLITERAL ( ds: -- fs: n -- )
10|  COMPILE FLIT  fpsc--  _fpsaddr @ HERE 4 COPYW
11|  8 ALLOT ( advance HERE past the fp value) ; IMMEDIATE
12|: >F ( ds: -- fs: -- n) STATE @ IF COMPILE FLIT  str>fpstr
13|  fpstr>here 8 ALLOT ELSE str>fpstr fpstr>fp THEN ; IMMEDIATE
14|: TRUNC (  F: x -- trunc[x] ) FLOOR  ndf0< IF >F 1 F+ THEN ;
15|: CEIL  ( F: x -- ceil[x] ) FLOOR >F 1 F+ ;  -->
```

```
#59-----+----1----+----2----+----3----+----4----+----5----+----6---
00|: FRAC  ( F: x -- trunc[x] ) FDUP TRUNC F- ;
01|
02|.(   Floating-point comparison routines...)
03|: F=  ( -- t/f | a b -- ) %00100000 doComp 0> ;
04|: F0= ( -- t/f |   a -- ) fpsc-- _fpsaddr @ @ 0= ;
05|: F>  ( -- t/f | a b -- ) %01000000 doComp 0> ;
06|: F<  ( -- t/f | a b -- ) %01100000 doComp 0= ;
07|: F0< ( -- t/f |   a -- ) fpsc-- _fpsaddr @ @ 0< ;
08|
09|.(   Floating-point variable handling...)
10|: FVARIABLE  CREATE 8 ALLOT ;
11|: F! ( ds:address --  fs:n --) fpsc-- _fpsaddr @ SWAP 4 COPYW ;
12|: F@ ( ds:address --  fs: -- n ) _fpsaddr @ 4 COPYW  fpsc++ ;
13|-->
14|
15|

#60-----+----1----+----2----+----3----+----4----+----5----+----6---
00|.(   Floating-point constant handling...)
01|: FCONSTANT ( ds: --  fs: n -- ) CREATE 8 ALLOT
02|  fpsc-- _fpsaddr @ HERE 8 -  4 COPYW
03|  DOES> _fpsaddr @ 4 COPYW fpsc++ ;
04|: FVALUE ( ds: --  fs: n -- ) FCONSTANT ;
05|: (FTO) ( body -- ) fpsc-- _fpsaddr @ SWAP 4 COPYW ;
06|: FTO ( ds: --  fs: n -- ) \ set FP value  e.g: >F 3.141 FTO PI
07|  BL WORD FIND DROP >BODY STATE @ IF [COMPILE] LITERAL  COMPILE
08|  (FTO) ELSE (FTO) THEN ; IMMEDIATE
09|: (+FTO) ( body -- ) DUP   _fpsaddr @ 4 COPYW fpsc++
10|  F+   fpsc-- _fpsaddr @ SWAP 4 COPYW ;
11|: +FTO ( ds: --  fs: n -- )
12|  \ the fp value n is added to the fp FVALUE
13|  BL WORD FIND DROP >BODY STATE @ IF [COMPILE] LITERAL COMPILE
14|  (+FTO) ELSE (+FTO) THEN ; IMMEDIATE
15|-->

#61-----+----1----+----2----+----3----+----4----+----5----+----6---
00|.(   Floating-point display routines...)
01|: F. ( F: n -- ) \ display the top fp number in decimal
02|  fpsc-- _fpsaddr @   \ pop top of fp stack
03|  0 0 0       \ free format output (CNSopt1 CNSopt2 CNSopt3)
04|  $0012 goFPL \ convert fp number to string at _fpstr
05|  _fpstr COUNT TYPE ( display it) ;
06|: .FS ( -- ) \ non-destructively display contents of fp stack
07|  _fpsc @ 0> IF _fpstack _fpsc @ 0 DO DUP
08|  0 0 0 \ free format output (CNSopt1 CNSopt2 CNSopt3)
09|  $0012 goFPL _fpstr COUNT TYPE SPACE  8 +   LOOP DROP
10|  ." <--TOP" CR  ELSE ." Empty " THEN ;
11|: .F$ ( -- ) \ display fp stack in hex, 8 bytes/line
12|  _fpsc @ 0> IF TRUE ZEROS ! _fpstack DUP _fpsc @ 8 * + SWAP DO
13|  CR I 8 + I DO I @ $. 2 +LOOP 8 +LOOP ." <--TOP" CR FALSE
14|  ZEROS ! ELSE ." Empty " THEN ;
15|-->
```

```
#62-----+----1----+----2----+----3----+----4----+----5----+----6---
00|: F$. ( F: n -- ) \ display FP number on top of stack in R100
01|  fpsc-- TRUE ZEROS ! _fpsaddr @ DUP 8 + SWAP DO I @ $.
02|  2 +LOOP FALSE ZEROS ! ;
03|: S>FP ( ds: n --  fs: -- n ) \ convert single to fp
04|  _fpsaddr @ DUP -ROT fpsc++ ! $000F goFPL ;
05|: FP>S    ( ds: -- n  fs: n -- )
06|  fpsc-- _fpsaddr @ DUP $000E goFPL @ ;
07|
08|.(   Transcendental functions...)
09|: EXP ( f: x -- exp[x] ) _fpsaddr @ 8 - $0006 goFPL ;
10|: LOG ( f: x -- log[x] ) _fpsaddr @ 8 - $0007 goFPL ;
11|: SQRT ( f: x -- sqrt[x] ) _fpsaddr @ 8 - $0008 goFPL ;
12|: COS ( f: x -- cos[x] ) _fpsaddr @ 8 - $0009 goFPL ;
13|: SIN ( f: x -- sin[x] ) _fpsaddr @ 8 - $000A goFPL ;
14|: TAN ( f: x -- tan[x] ) _fpsaddr @ 8 - $000B goFPL ;
15|: ATN ( f: x -- atn[x] ) _fpsaddr @ 8 - $000C goFPL ;       -->

#63-----+----1----+----2----+----3----+----4----+----5----+----6---
00|: POW ( f: base exp --- base^exp )
01|  fpsc-- _fpsaddr @ DUP 8 - SWAP $0005 goFPL ;
02|>F 3.141592653590 FCONSTANT PI
03|
04|ffaihm @ h !
05|
06|>F 0.01745329251994 FCONSTANT RAD/DEG
07|>F 57.295779513082 FCONSTANT DEG/RAD
08|>F 2.718281828459 FCONSTANT EULER_E
09|: >RAD ( f: deg -- rad )  RAD/DEG F* ;
10|: >DEG ( f: rad -- deg )  DEG/RAD F* ;
11|: LOG10 ( f: x -- log10[x] ) LOG >F 0.43429448190325 F* ;
12|: EXP10 ( f: x -- 10^x ) >F 10 FSWAP POW ;                  -->
13|
14|
15|

#64-----+----1----+----2----+----3----+----4----+----5----+----6---
00|\ fixed format display routines:
01|\   *intLen places to left of decimal point, including sign
02|\   *fracLen places to right of decimal point, including decimal
03|\    point
04|\   *mask includes bits (sign, E-notation options) to be ORed
05|\   with CNSopt1 = 1
06|.(   Fixed format display routines...)
07|: fixFmt. ( ds:intLen fracLen mask --  fs: n -- )
08|    \ display the top fp number in fixed decimal format
09|    fpsc-- _fpsaddr @   \ pop top of fp stack and get address
10|    SWAP 1 OR \ fixed format output + sign & E-notation options
11|    3 ROLL 3 ROLL \ put in order:FPaddr1 CNSopt1 CNSopt2 CNSopt3
12|    $0012 goFPL \ convert fp number to string at _fpstr
13|    _fpstr COUNT TYPE  \ display it
14|;                                                            -->
15|
```

```
#65-----+----1----+----2----+----3----+----4----+----5----+----6---
00|\ Each of these routines formats fp value n and
01|\   calls fixFmt. to display it:
02|: FF. ( intLen fracLen -- ) ( F: n -- )
03|   0 fixFmt. ;  \ fixed format
04|: FF+. ( intLen fracLen -- ) ( F: n -- )
05|   %110 fixFmt. ;  \ fixed format with '+'
06|: FFE.  ( intLen fracLen -- ) ( F: n -- )
07|   %1000 fixFmt. ;  \ E-notation fixed format
08|: FFE+.   ( intLen fracLen -- ) ( F: n -- )
09|   %1110 fixFmt. ;  \ E-notation fixed format with '+'
10|: FFX. ( intLen fracLen -- ) ( F: n -- )
11|   %11000 fixFmt. ;  \ extended E-notation fixed format
12|: FFX+.   ( intLen fracLen -- ) ( F: n -- )
13|   %11110 fixFmt. ;  \ ext. E-notation fixed format with '+'
14|
15|.( Floating-point library loaded.)
```

23 ASM>CODE Utility

The **CODE:** equivalents of the assembly language words were produced automatically with a small utility written specifically for the development of the double-precision library.

ASM>CODE (pronounced "assembly to code") takes a *pre-loaded* assembly language word, as assembled by the TurboForth assembler, and produces a **CODE:** equivalent word in a DV80 disk file specified by the user.

23.1 ASM>CODE syntax

The syntax for the **ASM>CODE** utility is very simple:

```
ASM>CODE <word-name> <file>
```

Where **<word-name>** is the name of a word in memory, assembled with the TurboForth assembler, and **<file>** is any valid file name. For example:

```
ASM>CODE D1+ DSK1.D1PLUS
```

Would produce a text file on DSK1 called D1PLUS that contained the following:

```
CODE: D1+
05A4 FFFE 1701 0594 ;CODE
```

Note that **ASM>CODE** opens the output file in append mode, so it is possible to (for example) create a blocks file containing many **ASM>CODE** statements (*i.e.*, a script) and have them directed cleanly and simply to the same text file.

The source code for **ASM>CODE** is as follows. Two disk blocks are required:

23.2 ASM>CODE Source Code

The following code is also downloadable from the TurboForth website as part of the utilities disk download.

```
#29-----+----1----+----2----+----3----+----4----+----5----+----6---
00|.( Loading ASM>CODE)
01|FORGET CFA      0 VALUE CFA     0 VALUE FNADDR  0 VALUE FNLEN
02|0 VALUE STRPOS  0 VALUE CCOUNT  FBUF: FileOut
03|: OpenFile  HERE COUNT FileOut FILE   FileOut #OPEN ;
04|: ClearHERE  HERE 1+ 64 BL FILL   64 HERE C! ;
05|: SetFileName  ClearHERE  FNADDR HERE 1+ FNLEN CMOVE ;
06|: SetLen  FNLEN 7 + HERE C! ;
07|: SetAppend  SetFileName S" DV80AS" HERE 1+ FNLEN + 1+ SWAP
08|  CMOVE SetLen OpenFile ;
09|: SetSeq  SetFileName S" DV80SO" HERE 1+ FNLEN + 1+ SWAP CMOVE
10|  SetLen OpenFile ;  : Asm? CFA @ CFA 2+ = ;
11|: SetName  ClearHERE  S" CODE: " HERE 1+ SWAP CMOVE
12|  CFA >LINK 2+ DUP @ 15 AND SWAP 2+ SWAP HERE 7 + SWAP 0 DO
13|  OVER C@ OVER C! 1+ SWAP 1+ SWAP LOOP 2DROP ;
14|: FlushLine  HERE COUNT FileOut #PUT ABORT" Can't write to file"
15|  ClearHERE  0 TO STRPOS ; -->
```

```
#30-----+----1----+----2----+----3----+----4----+----5----+----6---
00|: PlaceCell  CFA @ N>S  STRPOS 5 * HERE 1+ + SWAP CMOVE
01|  1 +TO STRPOS  2 +TO CFA ;
02|: &;  S" ;CODE" STRPOS 5 * HERE 1+ + SWAP CMOVE ;
03|: ProcessWord  SetName FlushLine BASE @ >R   16 BASE !
04|  ZEROS @ >R   TRUE ZEROS !  UNSIGNED @ >R  TRUE UNSIGNED !
05|  2 +TO CFA  BEGIN  CFA @ $045C <> WHILE PlaceCell STRPOS 12 =
06|  IF FlushLine THEN REPEAT STRPOS 0> IF &; FlushLine ELSE CFA @
07|  $045C = IF &; FlushLine THEN THEN R> UNSIGNED !  R> ZEROS !
08|  R> BASE ! ; : ASM>CODE  CR  ' TO CFA   BL WORD TO FNLEN
09|  TO FNADDR  CFA IF Asm? IF SetAppend IF SetSeq
10|  ABORT" Cant open file" THEN ProcessWord FileOut #CLOSE ELSE
11|  TRUE   ABORT" Not an assembly language word" THEN ELSE TRUE
12|  ABORT" Word not found" THEN ;
13|.( ASM>CODE loaded.)
14|.( Usage: ASM>CODE <name> <file>)
15|.( E.g.: ASM>CODE EXPECT DSK1.EXPECT)
```

24 Questions, Bug Reports, *etc*.

The author of the Floating-point library, and the author of TurboForth are regular posters in the TI-99/4A Programming Forum at http://www.atariage.com

Additionally, TurboForth has a dedicated website with a newly opened Forum where support requests can be posted:

http://turboforth.net