# FORTH DIMENSIONS

VOLUME IV. NUMBER 1 $2.50

# INSIDE:

## FIXED-POINT MATH

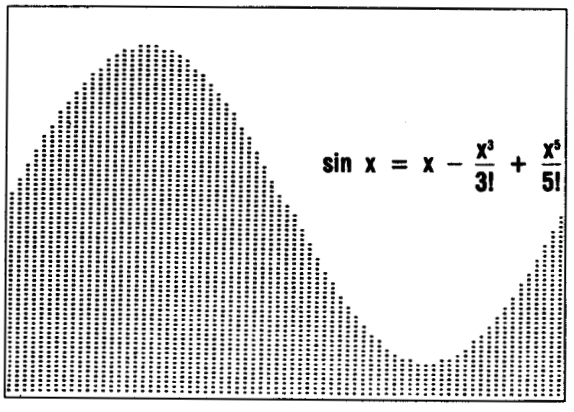
Fixed-Point Trig by Table Lookup ..... *John James* ..... 6
Fixed-Point Trig by Derivation ..... *J. Bumgarner* ..... 7
Fixed-Point Square Root ..... *Klaxön Suralis* ..... 9
Fractional Arithmetic ..... *Leo Brodie* ..... 13
The Cordic Algorithm for
Fixed-Point Polar Geometry ..... *Alan T. Furman* ..... 14
Quadruple Word Simple Arithmetic ..... *David A. Beers* ..... 17

## FLOATING-POINT MATH

High-Level Floating Point ..... *Michael Jesch* ..... 23
Vendor Support of Floating Point ..... 26

## DEPARTMENTS

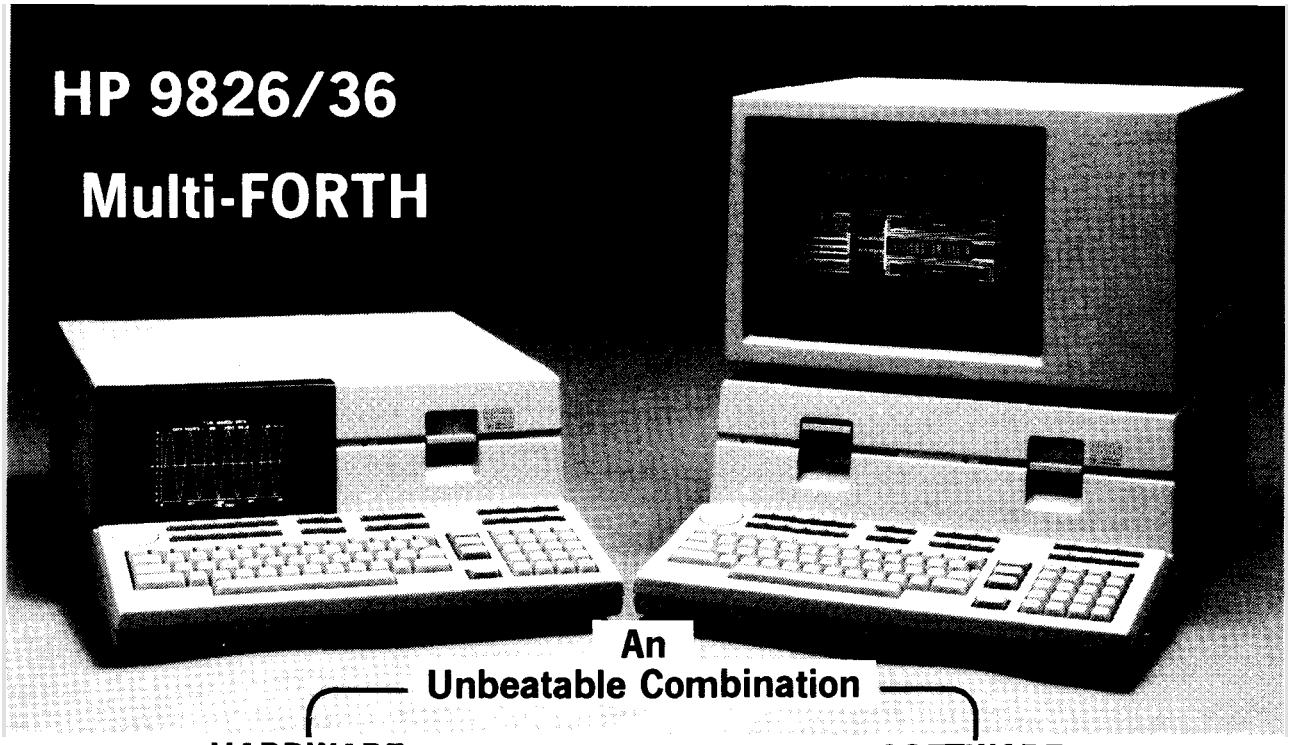
Letters ..... 1
Standards Corner: ..... *Robert L. Smith* ..... 3
New Proposals for Relationals
and Multiplication Operators
Techniques Tutorial: Defining Words ..... *Henry Laxen* ..... 4
New Product Announcements/Reviews ..... 30




# MATH EDITION

*Now Available On*

# HEWLETT PACKARD DESKTOP COMPUTERS




**An Unbeatable Combination**

**HARDWARE** **SOFTWARE**

The HP 9826A and 9836A are two of Hewlett-Packard's newest and most powerful desktop computers. Each is based on the Motorola MC68000 microprocessor. Both machines have full graphics capability and up to 2 full megabytes of user read/write memory. Both operate on 5¼" flexible disc drives (the 9836A has two) which feature 264K bytes of mass storage. While the 9826A has an integral 7" (178mm) CRT which makes it useful for computer-aided testing (CAT) and control, the 9836A has a full 12.2" (310mm) CRT which makes it ideal for computer-aided engineering (CAE) applications. Each model features the following:

- Seven levels of prioritized interrupt
- Memory-mapped I/O
- Built-in HP-IB interface
- Standard ASCII keyboard with numeric keypad and international language options
- Ten (20 with shift) user-definable soft keys with soft labels
- Rotary-control knob for cursor control, interrupt generation and analog simulations
- System clock and three timers
- Powerfail recovery option for protection against power lapses
- Seven additional interface cards
  - — DMA controller (up to 2.4 mb/sec)
  - — 8/16 bit bi-directional parallel
  - — Additional HPIB interface
  - — Serial RS232/449
  - — BCD
  - — Color video(RGB) 3 planes 512 x 512 8 color

## HP 9826/36 Multi-FORTH
## HP PRODUCT # 97030JA

Multi-FORTH was developed in 1979 by Creative Solutions, Inc. The standard product has been substantially modified to take full advantage of the 9826/36 hardware features.

**Multi-FORTH features**

- 79 standard programming environment
- Multitasking
- Full screen editor
- In-line structured assembler
- I/O and graphics extensions
- Loadable H.P. floating point (IEEE format)
- Extensive user manuals and documentation

**Optional Features:**

- Meta compiler
- Multi user
- Data access methods library

This product is part of HP PLUS — a program for locating user software. It has been developed by an independent software supplier to run on HP computer systems. It is eligible for HP PLUS as determined by references from satisfied end users. Support services are available only through the software supplier. Hewlett-Packard's responsibilities are described in the Responsibilities Statement below.

***Responsibilities Statement***

*HP PLUS software was developed by an independent software supplier for operation on HP computer systems. The supplier is solely responsible for its software and support services. HP is not the manufacturer or developer of such software or support. HP disclaims any and all liabilities for and makes no warranties, expressed or implied, with respect to this software. Distribution of this product or information concerning this product does not constitute endorsement of the product, the supplier, or support services.* **The customer is responsible for selection of the software it purchases.**

For more information, please write

**Marvel Ross, Hewlett Packard Company**

3404 East Harmony Road, Ft. Collins, CO. 80525

# Letters . . .

### Character Witness

Dear Fig,

Mr. Arthur Goldberg's letter in your February, 1982 issue is misleading. It implies that because only the first three characters and length of names are stored in headers, the user must type FORTH source code in an abbreviated, three character form. In fact, FORTH was one of the earliest languages to abandon abbreviations and emphasize typing fully spelled out words, for clarity, even though the internal storage is compressed to save space.

The problems with the three character system have to do only with its efficiency as a data compression scheme. There are two such problems. The first is the failure to distinguish similarly spelled names, forcing the user to select less than optimal nomenclature to avoid collisions.

The second is the inability to display the original full name from the dictionary. A presumption that names are used only in source code is thereby implied. This is probably true when used by a programmer, but when high-level, job-oriented languages are developed in FORTH, the nonprogrammer may be putting new data structures into the dictionary interactively. He may then want to see later what was put in, and abbreviations would not be acceptable. This second objection, by the way, applies to MMSFORTH's extension of the three character scheme to include a hash value to improve discrimination among names while conserving space.

Of course, keeping the full name comes at the expense of space. This can be very important, as illustrated by the article on **WORD** in your Dec. issue, where the programmers had to selectively delete headers in order to compile a 48K of code.

*George Lyons*
*Jersey City, New Jersey*

### De-California

Dear Fig,

I wholeheartedly applaude your efforts to "de-Californize" FORTH. I hope that someday FORTH will replace BASIC as the common language of computers. FORTH won't, as long as it remains a local phenomena.

*Eric Perrault*
*Seattle, Washington*

### Benchmark Ballyhoo

Dear Fig,

I'd like to shed some light on C.H. Ting's review of Timin FORTH. First, my computer system has the ususal 4 MHz clock, not 6 MHz as reported. Second, it was version 1.1 of Lab. Microsystems FORTH that was compared with mine.

Dr. Ting was quite precise in his timings. They apply to a 4 MHz system clock. At the time of the testing we did not know that Lab. Microsystems was releasing a version with improved execution speed. I have since run version 1.13 of their software and have found it to run at approzimately the same speed as Timin FORTH, except where disk block I/O is involved. The Timin system of 8 interleaved sectors per block results in much faster disk access. Here are two comparison tests that I ran. One of these does arithmetic and the other does disk I/O:

| Test | Timin FORTH rel. 2 | Z-80 FORTH ver. 1.13 |
|---|---|---|
| 1200 divides/ multiplies | 4.3 sec. | 4.2 sec. |
| CLEAR screens 12 thru 24 | 7.9 sec. | 36.1 sec. |

The disking results are system dependent.

*Mitchell E. Timin*
*Timin Engineering Company*
*San Diego, California*

Dear Fig,

Thank you for the opportunity to comment on Mr. Timin's letter. As usual, Mr. Timin's timings are obsolete long before they are published. We have been shipping Z-80 FORTH version 1.14 since July of 1981; this revision executes about 20% faster than the one Mr. Timin tested. It performs the Interface Age primes benchmark in 108 seconds on a 4 MHz CPU, while 8080 fig-FORTH completes the same benchmark in 157 seconds at the same clock speed.

It is quite true that disk access in a fig-FORTH system is much faster than in Z-80 FORTH. Fig-FORTH writes screens directly to the diskette, while Z-80 FORTH maps screens onto a standard operating system random access disk file. The latter method permits the software to be completely independent of the size and density of the disk media, and allows FORTH programs to coexist with other types of utilities and language processors. If the number of systems sold is any indication, our approach to disk management has won vastly more user acceptance than Timin's FORTH. In addition, Mr. Timin forgot to mention that both his company and Micromotion recently announced new FORTH packages that use operating system files for screen storage just like Z-80 FORTH.

Although I felt obligated to answer the points raised by Mr. Timin, I think that interchanges of letters along these lines could probably go on forever and do not really serve any useful purpose. If Mr. Timin wants publicity for his products, he can pay for his advertising like the rest of us. You might find it interesting to know that although Mr. Timin has felt quite free to publish timings and comments about Z-80 FORTH, he is not a registered owner of any of our software.

*Ray Duncan*
*Laboratory Microsystems*
*Los Angeles, California*

***From the Editor***

**I'd like to thank all the authors who contributed to this issue. The quality of their work is outstanding. I'd also like to thank Timothy Huang and Peter Helmers who wrote excellent articles for which there was simply no room. (Their articles will eventually appear, either here or in one of the conference proceedings.)**

**Next issue's theme is "Hardware Control," including process control, robotics, etc. (See box on pg. 8 for future themes.)**

**Keep writing!**

***Leo Brodie***


**FORTH Dimensions**

Published by FORTH Interest Group

Volume IV, No. 1 May/June 1982

Printing/Circulation/Advertising
Roy C. Martens

Editorial/Production
Leo Brodie

FORTH Dimensions solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. Unless noted otherwise, material published by the FORTH Interest Group is in the public domain. Such material may be reproduced with credit given to the author and the FORTH Interest Group.

Subscription to FORTH Dimensions is free with membership in the FORTH Interest Group at $15.00 per year ($27.00 foreign air). For membership, change of address and/or to submit material, the address is: FORTH Interest Group, P.O. Box 1105, San Carlos, CA 94070

# GET IT UP QUICKER

**NOW**
there's a revolutionary new data base manager and operating system with an easy to use conversational language.

This is the first truly relational system with full conversational JOIN capabilities that is available on a microcomputer.

**FORTH BASED SYSTEM,**
*not only written in FORTH, but the* user programming language is an extended subset of FORTH.

**EASY TO USE**
interactive commands to set up Data files and Fields which may be CHANGED without altering any application work.

**QUERY LANGUAGE**
to ADD, CHANGE, DELETE, LIST or JOIN files or data. Reports, including analytical and exception reports, can be generated immediately and often with a simple one line statement.

**POWERFUL**
Data Manipulation language, an extended subset of FORTH, lets you quickly write customized reports and other programs.

**FULL SCREEN EDITOR**
which helps you store, write and change programs or procedures for use later or simply to store and correct text for letters or memos.

**AMAZING PERFORMANCE AND FLEXIBILITY**
because the system is written in FORTH.

**TRS/80 Models I or III**
Lists or quantity ONE prices are:

| | |
|---|---|
| **DATA ACE√** User Guide | $30.00 |
| MMSFORTH | $125.00 |
| **DATA ACE√DBM** - database accesses, data definition language, editor and catalog and User Guide - | $150.00 |
| Query language - | $100.00 |

(MMSFORTH is a prerequisite for **DATA ACE√** (**DATA ACE√DBM** is a prerequisite for **DATA ACE√** Query language.)

Existing MMSFORTH users may obtain the complete **DATA ACE√** package for $250. Those using MMSFORTH for the first time will be investing $375.00.

(MMSFORTH IS A TRADE MARK OF MILLER COMPUTER SERVICES).

**TRS/80 Model II**
For $75 you get a User Guide and a demonstration disk. Once you know what you want, you may choose the exact package for your needs. **DATA ACE√** is also available for hard disk and with application start up kits.

**DATA ACE√ is a smart and completely integrated system** - operating system, relational data base manager, query language, structured programming language, full screen editor, catalog, application packages and commitment to future development.

**DATA ACE√**

## YES– show me more / send me

- ☐ the DATA ACE Model II User Guide and demonstration disk $75.
- ☐ the DATA ACE – Model I/III User Guide $30.
- ☐ a copy of MMSFORTH $125
- ☐ a copy of DATA ACE DBM $150
- ☐ a copy of DATA ACE Query Language $100.

Enclosed is my check/credit card details. (We accept Visa or M/C)

card no. _____ exp. / /

NAME _____

COMPANY _____

ADDRESS _____

CITY _____

STATE _____ ZIP _____

TELEPHONE _____

**COMPUTER SOFTWARE DESIGN INC**
FORMERLY AREGON SYSTEMS INC.

*1911 WRIGHT CIRCLE*
*ANAHEIM, CA., 92806*
*714/634 9012*

# Proposals for Relationals and Multiplication Operators

By ROBERT L. SMITH

**Relationals.** Traditionally FORTH has had very simple primatives for comparing arithmetic values. Usually the overflow conditions have been ignored, leading to occasional surprises. Consider, for example, the traditional definition of < (which is not 79-Standard):

```
: <   - 0< ;
```

In this example the top element on the stack is subtracted from the second element. The sign bit of the result is examined to check for a negative result. If the result is negative, the result is replaced by the value 1, otherwise it is replaced by zero. Since the arithmetic is done on a 16 bit basis, with no regard for Carry or Overflow bits, we find with the above definition that

```
-20000 20000 <
```

will return the value 0, meaning false! At the Rochester FORTH Standards Conference, Brodie and Sanderson demonstrated that this result is to be expected if we consider numbers to lie on a "number circle" which has no ending point, but an overlapping of numbers such that −16K is the same as +48K (K = 1024).

The FORTH Standard specifies that the comparison on two numbers using the word < is to be a true arithmetic comparison. In this case the numbers are considered to lie on a line from −32768 to +32767. The most obvious way to accomplish the arithmetic test is in a code definition, particularly if the machine has testable bits for Overflow, Zero, and Sign for arithmetic results. However, one of the most popular microcomputers, the 8080, does not generate an Overflow bit. It does however have a Carry bit, and this may be used in a fairly clever way. (This technique was recently rediscovered by Klaxon Suralis and presented at the Northern California FIG meeting). Consider first the comparison of unsigned numbers. The appropriate word is U<. In this case numbers are considered to lie in the range 0 to 65535. The best way to implement U< is in a code word in which the two numbers are subtracted and the carry bit is examined, and the final value of 0 or 1 is set accordingly. Note that the actual state of a carry bit depends on the machine being used. Some machines use the carry bit to mean "borrow" when a subtraction is made. Others merely complement the subtrahand and add, with the resulting carry being the same as in normal addition.

For the purposes of arithmetic comparison, one can make a translation from the signed number line to the unsigned number line by reversing the sign bits of the two operands. Suralis suggests calling this operation 2FLIP. This is best done in code, but a simple high level definition is as follows:

```
DECIMAL  16 BASE !
( Make the base
 Hexadecimal )
: 2FLIP   8000 XOR
SWAP  8000 XOR SWAP ;
```

The XOR operation could be replaced by + if that is faster. The definition of a 79-Standard " < " could then be:

```
: <   2FLIP U< ;
```

This is probably the best technique for implementing < on the 8080. You should try some examples to convince yourself that this technique actually works.

Suralis also suggests a related technique to convert unsigned address parameters to signed values to be used with the 79-Standard DO-LOOPS in a transportable manner. The loop indices are then converted back to the desired address value by applying an additional FLIP. It is an interesting method, but I think that it would be best if the Standard could be changed to reflect the need for unsigned loops (See the previous Standards Corner for one suggestion).

**Signed and Unsigned Multiplication.** There may be some confusion over signed and unsigned multiply operations. Consider first the simple multiply operator * . This takes two single precision operands and produces a single precision result. There is a potential overflow problem in this case since the product of two 16 bit quantities potentially may produce a 32 bit result. However, if the result can be represented properly in 16 bits, the correct result will be obtained whether we consider the operation to work on signed or unsigned operands!

The FORTH-79 unsigned multiply operator is U* . This is a mixed mode operator which returns an unsigned double number product of two unsigned single number operands. There are some additional multiply operators which are not in the Standard, but may be useful. If we wish the product of two double precision numbers, we can define a suitable word as follows:

```
: D*   ( dl d2 --- d3 )
 OVER  5 PICK  U*
 6 ROLL  4 ROLL  *  +
 2SWAP  *  + ;
```

The operands and results may be considered as either signed or unsigned double precision quantities, provided that the product is representable in 32 bits. In the above definition the Double Number Word Set extensions of FORTH-79 are assumed.

In fig-FORTH there is a mixed mode multiplication named M* . M* takes 2 signed single precision numbers and produces a signed double number product. There are a number of ways of defining M* . The following has the advantage of defining another fig-FORTH word.

```
: S->D   ( n --- d )
 DUP 0<
 IF  -1  ELSE  0  THEN ;

: M*   ( n1 n2 --- d )
  >R  S->D  R>     S->D  D* ;
```

The purpose of the word S->D is to sign extend a single number to a double number format.

# Defining Words

By HENRY LAXEN

This time we will take a look at the FORTH words **CREATE DOES>** and try to clear up all of the confusion surrounding them.

**CREATE DOES>** live on the third rung of the FORTH ladder. Let's go up it one step at a time. On the first rung is the Assembler. It allows you to create definitions out of the native machine instructions.

**:**, **CONSTANT**, **VARIABLE**, **VOCABULARY**, and **CREATE** live on the second rung. They allow you to create new FORTH words. Most systems, such as BASIC, FORTRAN, and PASCAL stop there. But in FORTH we are allowed to proceed to the third rung of the ladder, namely we have the ability to add new words to FORTH which are capable of defining yet other words.

Words that contain **CREATE DOES>** define other words. There is nothing mysterious about this; words that contain **CONSTANT**, **VARIABLE**, etc. do the same thing. The difference is that the only thing **CONSTANT** can do is define things that behave like constants; i.e., they push their value onto the parameter stack. Similarly the only thing words defined by **VARIABLE** can do is push the address of their data on the stack. With **CREATE DOES>** we have control over their run time behavior. We can create new entities that behave in totally unique ways both at compile time and at run time.

Suppose I want a word similar to **SPACES**, except that instead of spaces it will print the given number of asterisks. And suppose I also want words to print backspaces and underscores, etc. I could copy the definition of **SPACES**, changing only the name and the character to be emitted each time. But this would be painful. What I really want to do is define a word that will define all these words. That's what **CREATE DOES>** is for. Let's define a defining word called **EMITTER** like this:

```
: EMITTER
   CREATE C,
   (remember the character)
   DOES>
     C@
     (fetch the character)
     SWAP
     ?DUP IF
     0 DO  DUP EMIT  LOOP
     (Run Time behavior)
   THEN DROP ;
32 EMITTER SPACES
42 EMITTER STARS
08 EMITTER BACKSPACES
95 EMITTER UNDERLINE
```

When **EMITTER** is executed, it **CREATE**s (compiles) a dictionary entry with that name. Next the **C,** takes the number from the stack and adds it to the dictionary as a single byte. Next **DOES>** executes, and for the time being just imagine that it stops compiling and fixes the word that was just defined, namely **SPACES**, **STARS** or whatever, so that when it is executed, the words between the **DOES>** and the **;** are executed. Thus **EMITTER** has created a new word, remembered the character to be sent, and specified the run time behavior of its child.

Now what happens when the child is executed? Well, **DOES>** is even trickier than we imagined. Besides determining the run time behavior of the child, it also manages to push the parameter field address of the child on the stack. *IMPORTANT!!* It pushes the parameter field address of the child, namely the pfa of **SPACES** or **STARS** or **BACKSPACES**. *Not* the parameter field address of **EMITTER**, the parent. This is really convenient, since we just happened to store the character that we wanted to send in the first byte of the parameter field with **C,** above. So the first thing we do is fetch it back with a **C@**. Now we have the count, which was there before, and the character, which we just fetched, on the stack. The rest of the code in the **DOES>** part is just what we need to do to send that character the right number of times.

Thus we have created a new class of words in FORTH, namely **EMITTER** words.

The applications of **CREATE DOES>** are not necessarily so trivial. We can define single and multidimensional arrays, trees, field and records; virtually any kind of data structure. Let's do one more example, and please imagine how you would attack the problem in say FORTRAN or PASCAL. The problem is the following: I want the computer to recognize some simple two word commands, and respond accordingly.

| **Verbs** | **Nouns** |
|---|---|
| TAKE | KEYS |
| GRAB | |
| DROP | LAMP |
| DISCARD | |
| EAT | FOOD |
| DRINK | WATER |

The responses are to consist of the following:

I've got the .....
I've thrown away the .....
Yummy, that was good .....
.....? Yuck, I think I've lost my appetite.
You're a little confused.

Before I proceed, I must add this warning. The version of FORTH I am using is the one described in the book *Starting Forth* by Leo Brodie. My **EXECUTE** operates on pfa's instead of cfa's, and my **'** is neither immediate nor state dependent.

Each of the definitions in Fig. 1 is called with the parameter field address of the calling word on the stack. **ID.** is a word that takes a name field address and prints the name. Next we must classify the verbs. The easiest way to do this is to make constants out of them, as we did in Fig. 2.

Now we define KEYS LAMP FOOD and WATER to print out the appropriate message (Figure 3).

Clearly we could duplicate this with LAMP FOOD and WATER, but there is an easier way. Suppose we define a defining word called ACTS-ON which expects four words to follow the word it is defining, namely the thing to do for taking, dropping, eating, or drinking. Figure 4 shows

how we could do this.

Now if we type TAKE FOOD it should respond with "I've got the FOOD," and if we say EAT KEYS it should respond "KEYS? Yuck, I think I've lost my appetite."

The above approach has many benefits over the previous infinite **IF ELSE THEN** approach. If we decide to add more verbs or objects, it will be much easier to change the above definitions than to go back and change all of those nested IF statements. Furthermore, the **CREATE DOES>** approach will be much more compact, and in this case much faster as well.

'Til next time, good luck, and may the FORTH be with you.



*Henry Laxen is part of Laxen & Harris Inc., a FORTH teaching and consulting company based in Hayward, California.*

**Fig. 1**

```
: .NAME     NFA ID.    ;
: GOT-EM    CR ." I've got the "    .NAME    ;
: GIVE-EM   CR ." I've thrown away the "    .NAME    ;
: YUMMY     CR ." Yummy, that was good "    .NAME    ;
: YUCK      CR .NAME ." ?  Yuck, I think I've lost my appetite." ;
: ALMOST    CR ." You're a little confused."  DROP     ;


0 CONSTANT TAKE
0 CONSTANT GRAB
1 CONSTANT DROP
1 CONSTANT DISCARD
2 CONSTANT EAT
3 CONSTANT DRINK
```

**Fig. 3**

```
: KEYS    ( n -- )
   [ LATEST PFA ] LITERAL SWAP
   DUP 0 = IF    DROP    GOT-EM    ELSE
   DUP 1 = IF    DROP    GIVE-EM   ELSE
   DUP 2 = IF    DROP    YUCK      ELSE
   DUP 3 = IF    DROP    YUCK      ELSE
      DROP ALMOST
   THEN THEN THEN THEN    ;
```

**Fig. 4**

```
: ACTS-ON
   CREATE
      ' ,    ' ,    ' ,    ' ,
   DOES>
      OVER 0 5 WITHIN IF    SWAP 2* OVER + @ EXECUTE
                      ELSE  ALMOST  DROP    THEN    ;

ACTS-ON KEYS    GOT-EM    GIVE-EM    YUCK       YUCK
ACTS-ON LAMP    GOT-EM    GIVE-EM    YUCK       YUCK
ACTS-ON FOOD    GOT-EM    GIVE-EM    YUMMY      ALMOST
ACTS-ON WATER   GOT-EM    GIVE-EM    ALMOST     YUMMY
```

# Fixed-Point Trig by Table-Lookup

By JOHN S. JAMES
Colon Systems · San Jose, California

Here is an easy way to get sine and cosine on your FORTH system, even if it does not have floating point. The precision is good enough for graphics and many other applications, and the routines are fast.

The principle is to use table lookup to return the sine of 0 through 90 degrees. Simple compuations can express the sine or cosine of any integer number of degrees, positive or negative, in terms of the sine of 0-90. Since only integers are available, the values returned are sine or cosine multiplied by 10,000.

Because the results are scaled up, you can use the FORTH scaling operation */ (multiply and then divide) to get the results you want. For example, to multiply a number on the stack by the sine of 15 degrees, use

**15 SIN 10000 */**

The word **TABLE** (defined in Block 32) is a general-purpose tool for creating singly-indexed tables of constants. The values to be compiled into the table are placed on the stack, followed by the number of those values; then **TABLE** is executed to create a table with the name which follows. In this example, we created the sine table in two parts, the sine of 51-90 degrees ( **SINTABLEA** ), and the sine of 0-50 degrees ( **SINTABLEB** ), because some FORTH systems have stack sizes less than 90 in order to take advantage of "zero page" for faster execution in some microprocessors, e.g. 6502. (If your system has a larger stack, as most do, making **SINTABLE** a table of all 91 values will both simplify the code and make it run faster.)

*[Another way to fill a table is to use a comma after each value. This approach eliminates the stack size problem, the need for a defining word with a DO ... LOOP, and the need to list the table arguments in reverse order. -Ed.]*

The code is believed to be FORTH-79 Standard (it was tested on a system still being developed, however). If you have fig-FORTH, change **CREATE** to **BUILDS>**, change **NEGATE** to **MINUS** , and remove **79-STANDARD** .

```
Screen 32
  0 ( TRIG LOOKUP ROUTINES - WITH SINE*10000 TABLE)
  1 79-STANDARD
  2 : TABLE ( VALUES N---. CREATE A TABLE)
  3    CREATE ( HEADER FOR THE TABLE)
  4    ( NUMBER OF VALUES) 0 DO , LOOP ( COMPILE THE VALUES)
  5    DOES> SWAP 2 * + @ ;    ( RETURN A VALUE)
  6
  7 ( CREATE A TABLE OF SINE OF 0-90 DEGREES.  THE TABLE IS
  8   CREATED IN TWO PARTS, 'SINTABLEA' AND 'SINTABLEB',
  9   SINCE SOME FORTH SYSTEMS ON 6502 HAVE LIMITED STACK.)
 10
 11 10000 9998 9994 9986 9976 9962 9945 9925 9903 9877
 12  9848 9816 9781 9744 9703 9659 9613 9563 9511 9455
 13  9397 9336 9272 9205 9135 9063 8988 8910 8829 8746
 14  8660 8572 8480 8387 8290 8192 8090 7986 7880 7771
 15 40 TABLE SINTABLEA


Screen 33
  0 ( TRIG LOOKUP ROUTINES)
  1
  2  7660 7547 7431 7314 7193 7071 6947 6820 6691 6561
  3  6428 6293 6157 6018 5878 5736 5592 5446 5299 5150
  4  5000 4848 4695 4540 4384 4226 4067 3907 3746 3584
  5  3420 3256 3090 2924 2756 2588 2419 2250 2079 1908
  6  1736 1564 1392 1219 1045 0872 0698 0523 0349 0175
  7  0000            51 TABLE SINTABLEB
  8 : SINTABLE ( 0-90---SINE*10000)
  9    ( COMBINE THE TWO SMALL SINE TABLES)
 10    DUP 50 >
 11    IF 51 - SINTABLEA ELSE SINTABLEB THEN ;
 12 : S180 ( DEGREES---SINE*10000. SINE OF 0-180)
 13    DUP 90 > ( 91-180 DEGREES?)
 14    IF 180 SWAP - THEN ( REFLECT)
 15    SINTABLE ;


Screen 34
  0 ( TRIG LOOKUP ROUTINES)
  1
  2 : SIN ( DEGREES---SINE*10000)
  3    360 MOD ( DOESN'T CHANGE SINE VALUE)
  4    DUP 0< IF 360 + THEN ( HANDLE NEGATIVE ARGUMENT)
  5    DUP 180 >
  6    IF ( 181-359 DEGREES)
  7      180 - S180 NEGATE
  8    ELSE
  9      S180
 10    THEN ;
 11
 12 : COS ( DEGREES---COSINE*10000)
 13    360 MOD 90 + SIN ;
 14
 15
```

*The output of WAVE is featured on our cover.*

```
Screen 35
  0 ( EXAMPLE OF USE OF THE TRIG LOOKUP ROUTINES)
  1
  2 : PLOT ( ARG---. PRINT A LINE OF ASTERISKS)
  3    CR
  4    ( ARG) 0 DO 42 EMIT ( '*') LOOP ;
  5
  6 : WAVE ( ---. PLOT A SINE WAVE)
  7    360 0 DO
  8      I SIN 300 / 40 + ( SCALE IT) PLOT
  9    5 +LOOP ( PLOT EVERY FIFTH DEGREE)
 10    CR ;
```

# Fixed-Point Trig by Derivation

By J. BUMGARNER
Colon Systems · San Jose, California

Trigonometric functions are not a part of most FORTH systems and they are necessary to do any really useful graphics routines. I considered table lookup trig functions but while they are fast for the values stored in the table they are not general enough for my purposes. I chose to use a method to generate the desired trig function directly from an input angle. The method chosen is called the Taylor-Maclaurin Series.

All necessary trig functions can be derived from any one. I chose to implement the sine because the input angles that cause the sine function to generate outputs over its entire range are plus and minus numbers that fit well in scaled 16-bit signed integer arithmetic. The Taylor-Maclaurin Series for the sine is:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots$$

where x is some angle in radians (180 degrees = 3.14159 or pi radians). This series will generate increasingly accurate approximations for the sine when more terms are used. To get an idea of the errors here are some values for the series truncated at various terms and for an angle of pi/2 radians (90 degrees). The value of the sine at pi/2 radians is exactly one.

| Term | sin x (approx.) | Error |
|---|---|---|
| x | 1.57 | 57% |
| −x 3/3! | 0.92 | −8% |
| +x 5/5! | 1.005 | .5% |
| −x 7/7! | 0.9998 | −.02% |
| +x 9/9! | 1.000004 | .0004% |

It's easy to see that the maximum error goes down fairly quickly when you add terms. One problem is that the terms of the series are hard to compute with integer arithmetic as there are high powers of x and the factorials (5! is read 'five factorial' and is equal to 5·4·3·2·1 = 120) in the denominator get big fast. Perhaps doing some algebra will help.

If we factor out an x and expand the powers and factorials we see a pattern develop that looks promising:

$$\sin x = x \left( 1 - \frac{x^2}{2\cdot3} + \frac{x^2\cdot x^2}{2\cdot3\cdot4\cdot5} - \frac{x^2\cdot x^2\cdot x^2}{2\cdot3\cdot4\cdot5\cdot6\cdot7} + \ldots \right).$$

By factoring out $-x^2/2\cdot3$ from all but the first term we get:

$$\sin x = x \left( 1 - \frac{x^2}{2\cdot3} \left( 1 - \frac{x^2}{4\cdot5} + \frac{x^2\cdot x^2}{4\cdot5\cdot6\cdot7} - \ldots \right) \right)$$

and finally if we carry out this scheme two more steps and truncate the series at the x 9/9! term we have

$$\sin x \approx x \left( 1 - \frac{x^2}{6} \left( 1 - \frac{x^2}{20} \left( 1 - \frac{x^2}{42} \left( 1 - \frac{x^2}{72} \right) \right) \right) \right).$$

This nested form of the truncated series is convenient to use with scaled integer arithmetic, since the highest power of x is 2 and the nasty factorials are hidden. The form also is very nice for factoring in FORTH.

For a scaling factor I chose to use 10,000. This fits my needs and is nice for humans. Larger scaling factors can be used to improve precision with 16-bit signed arithmetic. A scale factor that is a power of 2 could be used to make the binary arithmetic more efficient. [See "Fractional Arithmetic" in this issue.] I was willing to sacrifice a little precision and speed to gain clarity and transportability.

The code for the sine series is shown in Block 36 (next page). The scaling factor and the square of x are used by name and not passed on the stack. The word **L** evaluates each nested term in the series and accumulates the result. Before using **L** in **(SIN)** I put a scaled 1 on the stack to start the accumulation. The final term in the series is explicitly evaluated last and uses the x value saved on the stack at the start of **(SIN)**.

In using scaled integer arithmetic you must remove a scale factor when you multiply two scaled values together. This is made very easy by FORTH's incredibly powerful word ***/** . The word **(SIN)** in this block can be used by itself if the input angles are scaled and restricted to the values shown in the comment on line 8. The compiled code occupies 99 bytes and takes on average of 61 ms to evaluate the sine on a 1 MHz 6502 system.

The words in Block 37 use **(SIN)** to obtain the three most useful trig functions for whole degree angles. **REDUCE** reduces any integer degree angle to a range acceptable to **(SIN)**. The words **COS** and **TAN** use trigonometric identities to develop the functions from **SIN** but are quite straightforward.

The tangent is infinite at plus or minus 90 degrees where the cosine is zero. Anything one does here is wrong, so I chose to do a simple thing **( SIN 3 * )** that at least tries and also preserves the sign of the tangent.

This block takes 185 bytes for a total of 284 bytes for the entire package. The average times of execution are 76 ms for **SIN**, 81 ms for **COS** and 164 ms for **TAN** — not fast but versatile and small. The words in Block 37 can be made to operate on fractions of a degree by multiplying the constants in **REDUCE** , **?MIRROR** and **COS** and the denominator only in **SIN** . For tenths of a degree multiply by 10, for 0.02 deg multiply by 50, etc. The limit is 91 (91*360 = 32760) but could be increased by clever programming in **REDUCE**, etc.

The accuracy of these functions is good even without any special precautions about rounding. The peak error of **SIN** as shown below is 0.02%. The test identity of $SIN^2 + COS^2$ should always be exactly one (10000 when scaled) and so shows the square of the error directly.

*(Source screens on next page)*

## Trig by Derivation (continued)

```
BLOCK 36
    0  ( Scaled integer SIN function.                          job82mar31)
    1
    2  10000 CONSTANT 10K    ( The scaling constant.)
    3        VARIABLE XS     ( The square of the scaled angle.)
    4
    5  ( a b --- m    ! m=10000-ax*x/b ... a common term in the series)
    6  : L   XS @ SWAP / NEGATE   10K */ 10K + ;
    7
    8  ( theta --- 10000*SIN  ! -15708 < theta < 15708  radians*10000)
    9  : (SIN)   DUP ( save x)   DUP 10K */ XS ! ( save x*x)
   10    10K ( Put 1*10000 on stack to start series.)
   11    72 L  42 L  20 L  6 L  ( Compute the series terms.)
   12    10K */  ( Finaly multiply by the saved x.)  !   ;S
   13
   14    Note: 1/10000 of a radian is close to 0.0057 degree or about
   15          21 arc seconds ... a very small angle.

BLOCK 37
    0  ( SIN, COSINE and TANGENT for whole degree angles.  job82mar31)
    1
    2  ( theta --- theta  ! Reduce Y-axis symmetry. 0-180 to 0-90-0)
    3  : ?MIRROR   DUP 90 >  IF 180 SWAP -  THEN ;
    4
    5  ( theta (any) --- theta (-90 to 90)  !  Angle range reduction.)
    6  : REDUCE   360 MOD  DUP 0<  IF 360 +   THEN DUP 180 <
    7     IF ( 0-180) ?MIRROR  ELSE 180 - ?MIRROR NEGATE  THEN ;
    8
    9  ( theta --- 10000*SIN or COS  !  Any angle in whole degrees.)
   10  : SIN   REDUCE   17453 100 */ ( deg. to rad*10000)   (SIN) ;
   11  : COS   360 MOD ( prevents possible overflow)  90 SWAP -  SIN ;
   12
   13  ( theta --- 100*TAN  ! TAN set to +-30000 for +-90 degrees.)
   14  : TAN   DUP SIN SWAP COS  ?DUP IF 100 SWAP */  ELSE 3 *  THEN ;
   15    ;S
```

## Upcoming Issues

Here's the planned schedule of themes for the remaining issues of Volume IV, including the deadline for theme articles:

| | |
|---|---|
| **2 Hardware Control** | **—** |
| **3 Operating Systems** | **7/15** |
| **4 Coding for ROM** | **9/01** |
| **5 Business Applications** | **10/15** |
| **6 Teaching FORTH** | **12/15** |

The projected themes for Volume V are: Project Management, FORTH in the Arts, Serial Communications, Laboratory Workstations, The FORTH Environment, and Looking Back (FORTH History).

We need articles for the "Operating Systems" issue. Possibilities include: o/s comparisons, writing FORTH on top of another o/s, writing other operating systems in FORTH, and adding useful features of other operating systems to FORTH. Please get in touch if you're interested.

Leo Brodie

# Fixed Point Square Roots

By KLAXON SURALIS

As you learn the FORTH approach to problem solving, you are sold on the virtues of fixed-point arithmetic. Confidently armed with the neat little techniques for adapting integers to nearly any application, you set out to write a numeric-type program.

While translating the requisite formulae into postfix notation, however, you run into a radical sign — a square root. In vain, you try to sneak around it or tunnel underneath; there's no way to avoid it. What can you do? Pile K upon K of floating point routines into your dictionary, just so you can use the SQRT function? Give up and go back to BASIC?

This can happen to you, whether your field is statistics, electronics, graphics, or special relativity. Square roots are everywhere, in all kinds of famous equations. Of all the "irrational" functions in mathematics, this is the most common — and the simplest to implement.

**Loading the Screens.**

The program is listed as screens 222 through 224. Of course, you may put them anywhere your mass storage allows. All three screens load in base ten; all are FORTH-79 Standard.

If your FORTH system is nonstandard, the source code will require only minor modification. The FORTH-79 word **R@** must be replaced by **R** (in fig-FORTH) or **I** (in polyFORTH). You may have to code **1—** as a separate **1** and **—**.

Additionally, some systems may omit the FORTH-79 required words **D<** and **U<**. If yours is one of them, just define (with **BASE** set to **DECIMAL**):

```
: D<  ROT 2DUP =
  IF ROT ROT DNEGATE D+ 0<
  ELSE SWAP < SWAP DROP
  THEN SWAP DROP ;

: U<  32768 + SWAP 32768 + SWAP < ;
```

Make SURE 32000 −32000 < returns 0 on your system. If not, demand a refund.

**Trying it Out.**

With all screens loaded, enter:

```
169 0 SQRT .
```

This should print 13, which is the square root of 169. Note that we had to put an 0 on top of 169, since SQRT demands a double-precision operand. Most systems would have accepted 169. , taking the decimal point as a signal to extend the value to 32 bits. Thus, you could try:

```
480249. SQRT .
```

and see the result: 693. Omitting the decimal point from 480249 , however, would print a garbage value and possibly a "stack empty" message.

SQRT works only for integers, dropping any fractional component in the root. So, if you type:

```
3. SQRT .
```

the system will respond with 1 , even though you know the answer should be something like 1.7320508... . But after all, this is fixed point arithmetic.

However, you can take the square root of 3 million:

```
3000000. SQRT .
```

and sneak a peek at three more root digits: 1732 . This is the principle embodied in the word XX defined on screen 224. Type:

```
3 XX
```

and there you have 1.732 . XX will type the square root of any integer up to 4095 . Since it takes a single-precision argument, you don't have to type a decimal point. Try a few roots of your own.

If you compare XX's results to a scientific calculator's, you'll see that XX is always accurate to three decimal places, although it never rounds upward. Round-to-nearest, while possible, usually does not justify the added complexity.

**SQRT's 32-bit radicand makes it perfect for finding hypotenuses, standard deviations, RMS values, and lots of other "root-of-sum-of-squares" procedures.**

Now look at the definition of XX. Right off, it multiplies n by one million. Then, after calling SQRT, it performs a special numeric conversion, which puts the three low-order root digits on the fraction side of a decimal point (ASCII code 46).

This combination of scaling and output formatting is the standard FORTH technique for ersatz floating point. The only wrinkle with SQRT is:

> If you multiply SQRT's radicand by a scale factor s, the root will emerge scaled by the square root of s.

More concretely, you've seen how XX shifts the radicand left six decimal digits, while the root is offset by only three places. Square root of 1,000,000 = 1,000. Get it?

**Using SQRT in Your Application.**

XX is included merely for demonstration purposes. In normal use, SQRT is the only word the rest of your application will need to know. It will work for any double number you feed it, and harbors no surprises or special cases (as far as I can tell).

SQRT's 32-bit radicand makes it perfect for finding hypotenuses, standard deviations, RMS values, and lots of other "root-of-sum-of-squares" procedures. If the input data are all like-scaled 16-bit integers, simply square them with **M*** or **U*** and accumulate them using **D+** (or **D−** if you need subtractions). Apply SQRT to this double-precision sum, and the root will be a 16-bit integer with the same scale factor as your original data.

Of course, it's up to you to ensure that **D+** won't overflow and that **D−** doesn't hand SQRT a negative number.

Since SQRT takes an unsigned radicand, your application is free to handle imaginary roots as appropriate. If you like, you may install simple error checking for negative radicands, or integrate SQRT into a complete fixed-point (!) complex numbers package.

**But How Does It Work?**

On big machines, square roots are extracted by a technique from calculus called "Newton's Method." It is best suited to CPUs with full floating-point arithmetic hardware.

The alternative approach, used in this SQRT, works by addition, subtraction, and shifting. The result is constructed bit by bit, in a fashion quite similar to classical binary long division.

As in quotient generation, we set up

*Continued on next page*

a partial remainder (initially equal to the radicand) and proceed to chip away at it. If we take away too much, we put it back and shift a "0" into the root; otherwise, the root gets a "1". Sixteen such trials take place, one for each bit of the root.

What's different is the quantity subtracted/added to that remainder. Instead of an unchanging divisor, we must use the root itself — as many bits of it as are already determined — with a binary "01" stuck on the end. As the calculation advances, this subtrahend gets wider and wider.

For all but the last two trials, 16-bit addition and subtraction are wide enough to cover the action. This 87.5% share of the job is done by EASY-BITS (screen 223). As it churns along, EASY-BITS uses left shifts to keep root and remainder aligned under the optimal 16-bit window.

Within EASY-BITS, the phrases " 2* 1– " and " 2* 3+ " may bewilder you. They are sneaky, optimized equivalents for " 1– 2* 1+ " and " 1+ 2* 1+ ", respectively. In two brief steps, they shift a new bit into the root and reestablish the "01" suffix.

Now, if you look at the definition of SQRT (screen 224), you'll see the 14 easy bits extracted in two chunks: first eight bits, then six more. There is a very good reason for this: it saves us from defining and using a 48-bit shifting operator, which would run slow as molasses.

You see, we needn't look at the radicand's low-order 16 bits until the root is half finished. At that point, the "ROT DROP" throws away a cell cleared to zeroes by "8 EASY-BITS". With the rest of the radicand thus exposed, we're ready to crank out six more root bits.

The last two bits are coaxed out by 2'S-BIT (screen 223) and 1'S-BIT

```
SCR# 222
0 ( SQUARE ROOT  simple auxiliary functions              SUR24MAR82)
1
2 FORTH DEFINITIONS DECIMAL      ( this is a 79-STANDARD program )
3
4 ( Your system may already include some or all of the following:)
5
6 : 2DROP   DROP DROP  ;
7 : 2DUP    OVER OVER  ;
8
9 : DU<   32768 +   ROT  32768 +  ROT ROT  D<  ;
10
11 : 2*    DUP +  ;          ( shift left 1 bit single precision )
12
13 : D2*   2DUP D+  ;        ( shift left 1 bit double precision )
14
15
```

```
SCR# 223
0 ( SQUARE ROOT  figuring easy bits & 1 hard one         SUR24MAR82)
1
2 : EASY-BITS (  drem1 partialroot1 count -- drem2  partialroot2 )
3         0 DO
4                 >R   D2* D2*                    ( shift drem twice )
5                 R@ -  DUP                    ( subtr. partial root )
6                 0< IF  R@ +   R> 2*  1-    ( restore drem & set 0)
7                    ELSE       R> 2*  3 +              ( or set 1 )
8        THEN ( proot shifted for next goround )  LOOP  ;
9
10 : 2'S-BIT  ( drem2  proot2 --  drem3  proot3  ; get penult. bit)
11    >R  D2*   DUP 0< IF      D2*  R@ -   R>  1+          ( set 1 )
12                     ELSE    D2*  R@ 2DUP
13                             U< IF   DROP  R>  1-        ( set 0 )
14                                ELSE  -    R>  1+        ( set 1 )
15                     THEN THEN  ;
```

```
SCR# 224
0 ( SQUARE ROOT  get last bit & put it all together     SUR26MAR82)
1
2 : 1'S-BIT   ( drem3 proot3 --  fullroot   ; remainder lost )
3   >R  DUP 0< IF    2DROP  R> 1+
4              ELSE D2*    32768 R@
5                   DU< 0= R> +  THEN   ;
6
7 : SQRT  (  ud1 -- u2   ;32-bit unsigned radicand--> 16-bit root)
8     0  1   8 EASY-BITS      ROT DROP   6 EASY-BITS
9     2'S-BIT   1'S-BIT  ;
10
11 ( and now, the SCENIC ROOT-- pure fun and easy to type )
12
13 : XX   ( n --   ;print sqrt of n to 3 decimal places. n <=4095 )
14     16 *   62500 U*      ( mult. n by one million in 2 stages  )
15     SQRT  0  <#   # # #   46 HOLD   #S   #>  TYPE SPACE  ;
```

(screen 224). As usual in computer arithmetic, it's the down-to-the-last-bit accuracy that really costs. The problem here is that the remainder and root become too wide for plain old + , − , and 0<. We have to worry about overflow.

Both 2'S-BIT and 1'S-BIT use " DUP 0<IF " to test a high bit before D2* shifts it into oblivion. Moreover, since " − 0<" can no longer be trusted, we must resort to unsigned comparisons ( U< and DU< ). These definitions wouldn't be so ugly if FORTH had the "carry bit" found on most microprocessors.

The principles of SQRT may be applied to roots of greater precision — 32 bits, frinstance. In fact, you could use this technique in a program to extract the square roots of floating point numbers!

**You Demand Proof?**

A formal derivation of SQRT's algorithm would waste a lot of FIG's paper on something nobody would read. So, I'll supply a hint, if you supply your own paper. Let:

N = the 32-bit radicand
r = the 16-bit root under construction, initially zero
b = the binary place value of the root bit to be found (initially 32768, always a power of 2)

Then, for each value of b from 32768 down to 1, DO:

```
IF  (r+b)²  <=  N
THEN  add b to r ;
```

In this approach, b is the only quantity that gets shifted. Now, simple algebra rewrites the condition as:

```
b (2r+b)  <=  N − r²
```

The changing value $N - r^2$ is our partial remainder. Note that the comparison is 32 bits wide, remember that multiplying by 2 and b are simple binary left shifts, and the rest is mere optimization.

Alternatively, you may find this algorithm described in any thorough treatment of computer arithmetic.

**Not Fast Enough?**

If SQRT runs too slow for your application, there's not much you can do without delving into machine CODE. If you don't need full 16-bit accuracy in your roots, you can substitute:

```
>R 2DROP R> 1−
```

for the final " 2'S-BIT 1'S-BIT " in the definition of SQRT. The two low-order root bits will then always be zero. The payoff: roughly 10% faster execution, depending on your system. Perhaps more importantly, you can throw out DU<, 2'S-BIT , and 1'S-BIT, cutting the program size in half.

If you have a FORTH assembler, the first thing you should try is defining 2* and D2* in low level. On my homebrew 6809 FORTH, this, by itself, nearly doubles execution speed. Actual mileage may vary, yours will probably be less.

Beyond this, all you can do is translate the whole mess into CODE. This task is straightforward, but, of course, CPU dependent.

Of those high-level programming languages which give you a choice, most make you accept a lot of unnecessary garbage just to get one function you really want. Floating point arithmetic and function packages serve as cases in point.

FORTH exhibits the opposite attitude. It lets you order a la carte, as this fixed-point square root program demonstrates. Such freedom and efficiency, however, cannot be divorced from the added responsibility of knowing exactly what you're doing and exactly what you want.

Good night and good luck.

# LEARN FORTH

QUICKLY, EASILY AND ECONOMICALLY

## IN JUST A FEW DAYS

FROM THE COUNTRY'S LEADING FORTH LANGUAGE INSTRUCTION FIRM

## Choose the FORTH Language Course(s) you need:

**CONSIDERING FORTH**
Course #101 • Describes how to determine if the FORTH language is appropriate for your project and company. Covers: advantages and disadvantages; planning; training; and project management considerations. 1 day of instruction. Cost $150*

**FORTH LITERACY CLASS**
Course #102 • "FORTH as a Second Language." An introduction to the FORTH language. Requires familiarity with at least one other computer language. 2 days of instruction. Cost $300*

**FORTH APPLICATIONS PROGRAMMING WORKSHOP**
Course #203 • Continuation of Course #102. Builds working knowledge of the FORTH language suitable to beginning applications programming. Requires familiarity with FORTH fundamentals. 3 days of instruction. Cost $450*

**FORTH APPLICATIONS DESIGN LECTURES**
Course #263 • Recommended for senior programmers and project managers. Purpose: shows how to develop quality applications using FORTH. Covers: management; standards; design and analysis tools; and evaluation techniques. 3 days of instruction. Cost $450*

**FORTH SOFTWARE DEVELOPMENT WORKSHOP**
Course #275 • A combination of Course #203 and Course #263 at an accelerated pace. 5 days of instruction. Cost $750*

**FORTH PROCESS CONTROL WORKSHOP**
Course #314 • Using FORTH in process control applications. Covers: design, programming, interrupts, producing ROMs, and testing. Requires familiarity with process control programming and FORTH applications programming. 4 days of instruction. Cost $600*

**FORTH BUSINESS PROGRAMMING WORKSHOP**
Course #324 • Using FORTH in transaction processing; data base applications; and text processing. Covers: design, standards, modifiability, maintainability, debugging and documentation. Requires familiarity with business applications and FORTH applications programming. 4 days of instruction. Cost $600*

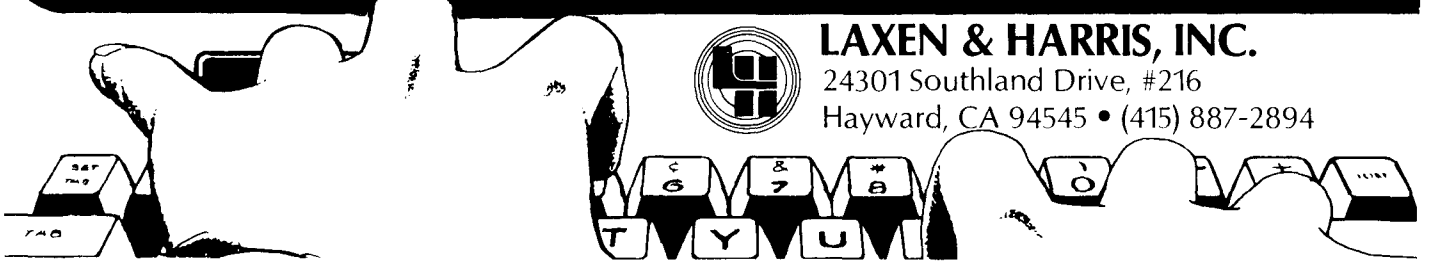
**FORTH SCIENTIFIC PROGRAMMING WORKSHOP**
Course #334 • Using FORTH for mathematical calculations; scientific functions; floating point; and laboratory instrument control. Requires familiarity with scientific applications and FORTH applications programming. 4 days of instruction. Cost $600*

**FORTH SYSTEMS PROGRAMMING WORKSHOP**
Course #344 • Extending, modifying and generating FORTH systems. Covers: internals, software tools, and metacompilation. Requires familiarity with system level programming and FORTH applications programming. 4 days of instruction. Cost $600*

*Class cost may vary depending on location and facilities and are subject to change without notice.

ALL COURSES INCLUDE TEXTS AND WORKBOOKS AVAILABLE NOWHERE ELSE

Classes are now being scheduled throughout the United States.
For more information without cost or obligation write or call:




**LAXEN & HARRIS, INC.**
24301 Southland Drive, #216
Hayward, CA 94545 • (415) 887-2894

# Fractional Arithmetic

By LEO BRODIE

The choice of fixed-point arithmetic over floating point arithmetic does not mean you're stuck with nothing but integers. There are good ways to express fractional values without paying the penalty for floating them.

One way to express a non-integer value is as the ratio of two integers; e.g., .66666 can be represented by the integers 2 and 3, where 2 is divided by three. You can express any fraction with very good accuracy by picking the right two integers. This kind of representation is called "rational approximation" (Chapter 5 of Starting FORTH).

You can also express fractions as single integers. All you have to do is assume that the integer has been scaled by some constant value. In John Bumgarner's article on fixed-point trig in this issue, he represents sines and cosines (which lie in the range between one and negative one) as integers scaled by 10,000. Another method that allows greater accuracy and the potential for greater execution speed scales the integer by 16,384.

In the listing below, we first define **+1** as a constant with the value 16384. In our scale, the integer 16384 represents the positive integer one. Remember that in binary, 16384 looks like this:

**0100000000000000**

In other words, the "1" is scaled up in binary such that there's an implied binary point just after the second most significant bit. Using 16384 as the scaling factor allows greater accuracy within 16 bits than does 10,000 (by a ratio of 16 to 10).

The next two words in the listing, ***.** and */.* , are the fractional math multiply and divide operators, written in high-level. Finally we have two words for input and output. **D–>F** converts a decimal number entered in the form x.xxxx (the digit left of the decimal point is optional) into our fractional representation. **.F** outputs a fraction in the form x.xxxx.

We could, for instance, multiply .75 by .5 in this fashion:

```
.7500 D–>F  .5000 D–>F  *.  .F
<ret> .3750 ok
```

Interestingly, if we ***.** a fraction times an integer, the result is an integer. For example:

```
28  .5000 D–>F  *.  . <ret> 14
```

With */** we can divide, say, –.3 by .95 like this:

```
–.3000 D–>F  .9500 D–>F  /.  .F
<ret>  –0.3157
```

We can also get a fraction result by using */.* on two integers. Thus

```
22 44 /. .F  <ret>  .5000
```

And we get an integer result if we */.* an integer by a fraction. To summarize, using the letter " f " for fraction and " i " for integer, here are the possible input/output combinations for ***.** and */.* :

```
f f *. f
f i *. i
i f *. i

f f /. f
i i /. f
i f /. i
```

It's no trick at all to add or subtract fractions — we just use + and –. For example, to solve this equation:

**7/34 + 23/99**

we can execute

```
7 34 /.  23 99 /.  +  .F
<ret> 0.4381
```

Besides allowing greater accuracy, using 16384 as the scale value allows you to define ***.** and */.* in assembler code extremely efficiently. (FORTH, Inc. uses this technique in their optional trig packages, and of course they use code-level definitions.)

Thanks to Greg Bailey for his thoughts on this topic.

```
Screen # 97
  0 ( fractional arithmetic)
  1 16384 CONSTANT +1
  2 : *. ( n n -- n)  +1  */ ;
  3 : /. ( n n -- n)  +1  SWAP  */ ;
  4 : D->F  ( d -- fraction) DROP  10000 /. ;
  5 : #.####  DUP ABS 0 <#  # # # #  46 HOLD # SIGN #> TYPE SPACE ;
  6 : .F  ( fraction -- ) 10000 *.  #.#### ;
  7
  8
  9
 10
 11
 12
 13
 14
 15
```

# The Cordic Algorithm for Fixed-Point Polar Geometry*

By ALAN T. FURMAN

A host of interesting shapes (circles, polygons, spoke patterns, rose curves, pinwheels) are most easily described in polar coordinates, whereas graphics devices (CRTs, plotters) are generally addressed via rectangular coordinates. Polar-to-rectangular conversion then is the key to many new possibilities.

The CORDIC algorithm (COordinate Rotation DIgital Computer) was originally invented in the 1950's for hard-wired implementation in a special-purpose navigation computer [Volder, J., IRE Trans. Electron. Comput. EC-8:330-334 (1959)]. It comes in two flavors: rectangular-to-polar conversion, and vector rotation by a given angle. The latter algorithm, which will be discussed here, will also do polar-to-rectangular conversion as a special case. I have found it quite handy for both CRT and plotter graphics.

**A host of interesting shapes are most easily described in polar coordinates, whereas graphics devices are generally addressed via rectangular coordinates. Polar-to-rectangular conversion then is the key to many new possibilities.**

I present the algorithm here (see box) in generalized form, both because

*Based on the author's presentation at the March 1982 meeting of FIGGRAPH.

ALGORITHM

Given: n-bit integers X, Y, and ANGLE, in twos-complement (MSB is sign)
Find: new X and Y, the original vector rotated CCW by ANGLE

```
begin
    X←X/K_n; Y←Y/K_n
    if ANGLE>0 then
        begin  ANGLE←ANGLE-2^(n-2); XLAST←Y; YLAST←-X   end
    else
        begin  ANGLE←ANGLE+2^(n-2); XLAST←Y; YLAST←-X   end;
    for i←0 step 1 until n-3
    do begin
        if ANGLE>0 then
        begin
            Y←YLAST+XLAST/(2^i);
            X←XLAST-YLAST/(2^i);
            ANGLE←ANGLE-α_i
        end
        else
        begin
            Y←YLAST-XLAST/(2^i);
            X←XLAST+YLAST/(2^i);
            ANGLE←ANGLE+α_i
        end;
        XLAST←X; YLAST←Y
    end
end
```

NOTES FOR THE ALGORITHM

1. The angle $\phi$ (in radians) is represented by the integer $\Phi$ defined as follows:

$$\Phi=-(-2^{n-1}\frac{\phi}{\pi})$$

For example, a straight angle is $-2^{n-1}$, $90^{o}$ is $2^{n-2}$, $-90^{o}$ is $-2^{n-2}$.

2. The $\alpha$'s are:

$$\alpha_0=2^{n-3}$$

$$\alpha_i=\mathrm{rnd}(\frac{2^{n-1}}{\pi}\tan^{-1}\frac{1}{2^i})$$

$$\alpha_{n-2}=1$$

3. The elongation factor $K_n$ is

$$K_n=\sqrt{\prod_{i=0}^{i=n-2}(1+2^{-2i})}$$

4. Loop limits are inclusive per ALGOL usage; that is, the loop is executed for i=n-3.

the original reference never actually does so, and because the fancy stack footwork (look, Ma — no variables) in my FORTH routine makes it hard to follow.

The main idea of the algorithm is this: we have an X,Y pair and an angle (let us assume it to be positive) by which the vector is to be rotated. The first go around the loop, we rotate the vector by a specially chosen angle $\alpha_0$. The resulting vector is not the exact rotated vector we seek, but is *closer* to it than the original vector. If we now subtract $\alpha_0$ from the given angle, we have the angle by which the new vector remains to be rotated. Similarly, we rotate by $\pm \alpha_i$ on the ith iteration. By the last iteration, the running angle converges to zero, and X and Y converge to the new vector.

What makes the algorithm efficient (especially when hardwired) is the way the angles $\alpha_i$ are chosen. Each rotation consists of adding to or subtracting from each component (X or Y) the other component divided by a power of 2 (that is, shifted). When this is done, the "rotations" also alter (increase) the vector's length — an unsolicited side effect. This alteration is not a problem, however, since the overall elongation factor $K_n$ is a constant for a given n.

In my Forth routine, I prescale the vector components by $1/K_n$, thereby allowing arguments over the full ±32767 range without risk of overflow.

The algorithm correctly handles twos-complement arguments and results automatically. The angle notation looks complicated, but it does have the nice property that overflows that occur when performing computations on these angles cause no actual errors. Example (16-bit arithmetic): the integer representing 120° is 21845. Doubling this number gives −21846, which represents −120°.

Some related routines:

```
: POLAR->RECTANGULAR
  ( radius 16bitANGLE -- Y X )
  0 ROT ROT ROTVECTOR ;

HEX
: PIRADIANS
  ( num denom -- 16bitANGLE )
  ( corresponding to π*num/denom radians )
  MINUS 8000 ROT ROT */ ;  DECIMAL

: DEGREES
  ( angle -- 16bitANGLE )
  ( angle in integral degrees )
  180 PIRADIANS ;
```

The apparent clumsiness of the 16-bit angle notation disappears through the magic of */. For example, to find the Cartesian coordinates of a point 1500 units from the origin on a ray rotated π/3 radians counterclockwise from the X-axis, one says

```
1500  1 3  PIRADIANS
POLAR->RECTANGULAR  . .
748 1298 OK
```

and so forth.

```
Screen # 28
  0 ( ROTVECTOR -- CORDIC VECTOR ROTATION   ATF MARCH 1982  1 OF 2 )
  1 ( 79-STANDARD FORTH )
  2 CREATE ALPHAS                          8192 ,  4836 ,
  3     2555 ,  1297 ,   651 ,   326 ,   163 ,    81 ,
  4       41 ,    20 ,    10 ,     5 ,     3 ,     1 ,
  5
  6 CREATE 2TOTHE                    HEX
  7        1 ,     2 ,     4 ,     8 ,    10 ,    20 ,
  8       40 ,    80 ,   100 ,   200 ,   400 ,   800 ,
  9     1000 ,  2000 ,  4000 ,  8000 ,                  DECIMAL
 10
 11 : RVSUB1
 12    >R ROT ROT OVER OVER R> 2 * 2TOTHE + @ / ;
 13
 14 : RVSUB2
 15    >R ROT ROT SWAP R> 2 * 2TOTHE + @ / ;


Screen # 29
  0 ( ROTVECTOR -- CORDIC VECTOR ROTATION              2 OF 2 )
  1 : RVSUB3
  2    >R ROT R> 2 * ALPHAS + @ ;
  3
  4 : ROTVECTOR ( y<old> x<old> angle -- y<new> x<new> )
  5    ROT 1002 1650 */ ROT 1002 1650 */ ROT
  6    DUP 0 > IF 16384 - ROT ROT SWAP NEGATE ROT
  7            ELSE 16384 + ROT ROT NEGATE SWAP ROT THEN
  8    14 0 DO
  9       I  OVER 0 > IF
 10       RVSUB1 + I RVSUB2 - I RVSUB3 -
 11       ELSE RVSUB1 - I RVSUB2 + I RVSUB3 + THEN
 12       LOOP  DROP ;
 13
 14 ( For FigFORTH, define CREATE as  0 VARIABLE  -2 ALLOT
 15     and  NEGATE  as  MINUS )
```

# Z-80® and 8086 FORTH

**FORTH Application Development Systems** including interpreter-compiler with virtual memory management, assembler, full screen editor, line editor, decompiler, demonstration programs, and utilities. Standard random access disk files used for screen storage. Extensions provided for access to all operating system functions. 120 page manual.

Z-80 FORTH for CP/M® 2.2 or MP/M ..... $ 50.00
8086 FORTH for CP/M-86 ..... $100.00
PC/FORTH for IBM® Personal Computer ..... $100.00

**Floating point extensions** for above systems. Specify software floating point, AMD 9511, AMD 9512, or Intel 8087 support.... additional $100.00

**Nautilus Cross Compiler** systems allow you to expand or modify the FORTH nucleus, recompile on a host computer for a different target computer, generate headerless code, and generate ROMable code with initialized variables. Supports forward referencing to any word or label. Produces load map, list of unresolved symbols, and executable image in RAM or disk file. (Prerequisite: one of the application development packages above for your host system)

Z-80 host: 8080 or Z-80 target ..... $200.00
Z-80 host: 8080, Z-80, or 8086 target ..... $300.00
8086 or PC/FORTH host: 8080, Z-80, or 8086 target..... $300.00

**FORTH Programming Aids** by Curry Associates. Includes Translator, Callfinder, Decompiler, and Subroutine Decompiler. 40 page manual. Used with Cross-Compiler to generate minimum size target applications. Specify Z-80 or 8086 FORTH screen file or fig-FORTH style diskette ..... $150.00

**Z-80 Machine Tests.** Memory, disk, console, and printer tests with all source code. Specify CP/M 2.2 or CP/M 1.4..... $ 50.00

**AMD-9511** arithmetic processor S-100 interface board.

Assembled and tested, without AMD 9511 ..... $200.00
Assembled and tested, with AMD 9511..... $350.00

PC/FORTH distributed on 5¼ inch soft sectored double density diskettes. All other software distributed on eight inch soft sectored single density diskettes. North Star and Micropolis formats available at extra charge.

Prices include shipping by UPS or first class mail within USA and Canada. Overseas orders add US $10.00 per package for air mail. California residents add appropriate sales tax. Purchase orders accepted at our discretion. No credit card orders.

Z-80 is a trademark of Zilog, Inc. IBM is a trademark of International Business Machines Corp. CP/M is a trademark of Digital Research, Inc.

**Laboratory Microsystems**
**4147 Beethoven Street**
**Los Angeles, CA 90066**
**(213) 306-7412**

# Quadruple Word Simple Arithmetic

By DAVID A. BEERS
Aregon Systems, Inc.

## Introduction

During the development of a property management application for the relational database system, DATA ACE, it became apparent that double word arithmetic with triple word intermediate products was occasionally giving incorrect answers. The incredible prices of California real estate necessitated ranges of millions to tens of millions of dollars and cents, and quite often a percentage return on these amounts was desired. This required more than a triple word intermediate product to retain all of the significant digits. The three possible solutions to our problem were quadruple word intermediate results, floating point, and prescaling. We felt that an unsophisticated programmer should be able to use DATA ACE without having to worry about the accuracy of intermediate results and this eliminated prescaling. Since we were dealing with money, we decided that the inaccuracy of floating point operations would not be tolerated. Thus Aregon decided that the answer for our particular situation was quadruple-word intermediate results.

The first crucial milestone of any project is that it work. This paper describes one method of implementing quadruple word simple arithmetic, though far from optimized, in high-level FORTH.

## Algorithm

My first attempt at quadruple word arithmetic followed the traditional BFBI approach (Brute Force and Bloody Ignorance). I implemented the addition and subtraction algorithms for the Z80 fairly well the first time. Once I had those, I merely did repetitive addition for the multiplication and repetitive subtraction for the division. Unfortunately, on values which actually utilized the range of a quadruple word number a single multiplication or division took as long as 5 seconds. This, obviously, was not satisfactory. I then discussed the algorithm possibilities with Kim Harris, who has done a considerable amount of work with large parallel processor design, and Dr. Jon Bentley, whose specialty is concrete complexity and algorithm design, and re-read the multi-word arithmetic sections of Knuth. The following algorithms are the result.

The addition and subtraction routines are merely multiple iterations of single word "add with carry" and "subtract with borrow" sequences to provide the full quadruple word range desired. Since the typical "add with carry" and "subtract with borrow" machine operations are not available in high level FORTH, the routines actually do two adds per iteration. The first addition adds the carry created by the previous loop iteration's addition, while the second addition actually adds the second operand. Since only one of the additions can result in a carry of one, the carries from these two additions are ORed together to provide the carry for the next iteration.

*Continued on next page*

```
 0 ( QUADRUPLE WORD ARITHMETIC   Q+ Q- SUPPORT WORDS  )
 1
 2 CREATE 1TEMP  8 ALLOT
 3 CREATE 2TEMP  8 ALLOT
 4
 5 : Q!    ( Q# ADDR - )
 6   DUP 8 + SWAP DO
 7     I !
 8   2 +LOOP   ;
 9
10 : Q@    ( ADDR - Q# )
11   DUP 6 + DO
12     I @
13   -2 +LOOP  ;
14
15
```

```
 0 ( QUADRUPLE WORD ARITHMETIC   Q+ Q- SUPPORT WORDS  )
 1
 2 : ?CARRY     ( #1 #2 RESULT# - BOOLEAN )
 3   -32768 AND ROT  -32768 AND ROT  -32768 AND ROT
 4   IF
 5     AND
 6   ELSE
 7     OR
 8   THEN  0= NOT  ;
 9
10 : ?BORROW    ( #1 #2 RESULT# - BOOLEAN )
11   ROT  ?CARRY  ;
12
13
14
15
```

```
 0 ( QUADRUPLE WORD ARITHMETIC   Q+ Q- ALGORITHM WORDS  )
 1
 2 : Q+TEMP      ( - )   ( ALL OPERANDS IN TEMP VARIABLES )
 3   0   0 6 DO
 4     1TEMP I + @  2DUP +  DUP 2SWAP ROT ?CARRY SWAP
 5     2TEMP I + @  2DUP +  DUP 2SWAP ROT ?CARRY
 6     SWAP 2TEMP I + !   OR
 7   -2 +LOOP    DROP   ;
 8
 9 : Q-TEMP      ( - )   ( ALL OPERANDS IN TEMP VARIABLES )
10   0   0 6 DO
11     1TEMP I + @ SWAP  2DUP -  DUP 2SWAP ROT ?BORROW SWAP
12     2TEMP I + @  2DUP -  DUP 2SWAP ROT ?BORROW
13     SWAP 2TEMP I + !   OR
14   -2 +LOOP    DROP   ;
15
```

*Listing continued on next page*

The multiplication routine examines from left to right each of the bits of the first operand. For each bit examined the accumulating result is first shifted left one bit. When a one bit is detected by the examination process, the current value of the second operand is added into the accumulating result. If the bit is zero, no addition is performed. When all of the bits have been examined, the accumulating result is complete.

Like the multiplication algorithm, the division algorithm uses powers of two to increase the speed by decreasing the number of iterative subtractions necessary to compute the quotient. It first multiplies the divisor by increasing powers of two until the resultant divisor is larger than the dividend. As it is doing this, it keeps track of the power of two involved. It then repetitively divides that resultant divisor and its associated power of two by two, and checks to see if the new resultant divisor is smaller than the current remainder (which on the first iteration is set equal to the dividend). If it is smaller, then the current divisor is subtracted from the current remainder and the current power is added to the current quotient. If the divisor is not smaller, then the subtraction from the remainder and the addition to the quotient are not done. When the power of two reaches zero the current value in the quotient and the remainder are the final results.

Both the multiplication and division routines initially compute the sign of the result and from then on work with the absolute values of the input operands. When a final absolute value result is arrived at, that computed sign is attached to the result and left on the stack.

The quadruple word numbers are stored in memory with the most significant word first and the least significant word last. The numbers, when placed on the stack, are placed with the least significant word deepest on the stack and the most significant word on the top of the stack. None of

```
 0 ( QUADRUPLE WORD ARITHMETIC   Q+ Q- DRIVING WORDS  )
 1
 2 : Q+      ( Q#1 Q#2 - Q#RESULT )
 3   1TEMP Q! 2TEMP Q!  Q+TEMP  2TEMP Q@  ;
 4
 5 : Q-      ( Q#1 Q#2 - Q#RESULT )
 6   2TEMP Q! 1TEMP Q!  Q-TEMP  2TEMP Q@  ;
 7
 8
 9
10
11
12
13
14
15


 0 ( QUADRUPLE WORD ARITHMETIC   Q* Q/ SUPPORT WORDS  )
 1
 2 : Q2*TEMP       ( - )   ( ALL OPERANDS IN TEMP VARIABLES )
 3   0   0 6 DO
 4     2TEMP I + @  DUP 0< SWAP  2*  ROT +  2TEMP I + !
 5   -2 +LOOP   DROP   ;
 6
 7 : Q2/TEMP       ( - )   ( ALL OPERANDS IN TEMP VARIABLES )
 8   2TEMP @ 0<
 9   8 0 DO
10     2TEMP I + @  DUP 1 AND  SWAP 2/
11     32767 AND  ROT IF
12       -32768 OR
13     THEN
14     2TEMP I + !
15   2 +LOOP   DROP  ;


 0 ( QUADRUPLE WORD ARITHMETIC   Q* Q/ SUPPORT WORDS  )
 1
 2 : Q2*     ( Q# - Q# )
 3   2TEMP Q!  Q2*TEMP   2TEMP Q@  ;
 4
 5 : Q2/     ( Q# - Q# )
 6   2TEMP Q!  Q2/TEMP   2TEMP Q@  ;
 7
 8 : Q<      ( Q#1 Q#2 - BOOLEAN )
 9   Q-  SWAP DROP SWAP DROP SWAP DROP  0<  ;
10
11 : Q0>      ( Q#1 - BOOLEAN )
12   DUP 0<  ROT ROT OR ROT OR ROT OR 0=   OR  NOT   ;
13
14
15


 0 ( QUADRUPLE WORD ARITHMETIC   Q* Q/ TEMPORARY VARIABLES  )
 1
 2 CREATE DIVISOR  8 ALLOT
 3 CREATE REMAINDER  8 ALLOT
 4 CREATE QUOTIENT  8 ALLOT
 5 CREATE POWER  8 ALLOT
 6
 7 CREATE RESULT  8 ALLOT
 8 CREATE OP2  8 ALLOT
 9
10
11
12
13
14
15
```

```
 0 ( QUADRUPLE WORD ARITHMETIC   UNSIGNED Q/ SUPPORT WORDS  )
 1 : @POWER      ( - )        ( ALL OPERANDS IN TEMP VARIABLES )
 2   BEGIN
 3     POWER Q@ Q2* POWER Q!     DIVISOR Q@ Q2* DIVISOR Q!
 4   REMAINDER Q@ DIVISOR Q@  Q<  END
 5   POWER Q@ Q2/ POWER Q!     DIVISOR Q@ Q2/ DIVISOR Q!  ;
 6
 7 : @QUOTIENT   ( - )        ( ALL OPERANDS IN TEMP VARIABLES )
 8   BEGIN
 9     REMAINDER Q@ DIVISOR Q@  Q<  NOT IF
10       REMAINDER Q@ DIVISOR Q@ Q- REMAINDER Q!
11       QUOTIENT Q@ POWER Q@ Q+ QUOTIENT Q!
12       THEN
13     POWER Q@ Q2/ POWER Q!  DIVISOR Q@ Q2/ DIVISOR Q!
14   POWER Q@ Q0> NOT END  ;
15


 0 ( QUADRUPLE WORD ARITHMETIC   UNSIGNED Q/  )
 1
 2 : |Q/| ( |Q.dividend| |Q.divisor| - |Q.remainder| |Q.quotient|)
 3   DIVISOR Q!  REMAINDER Q!
 4   0 0 0 0 QUOTIENT Q!  1 0 0 0 POWER Q!
 5   @POWER
 6   @QUOTIENT
 7   REMAINDER Q@  QUOTIENT Q@  ;
 8
 9
10
11
12
13
14
15


 0 ( QUADRUPLE WORD ARITHMETIC   UNSIGNED Q*  )
 1
 2 : |Q*|     ( D#1 D#2 - Q#1 )
 3   0 0 OP2 Q!  0 0   0 0 0 0 RESULT Q!
 4   63 0 DO
 5     RESULT Q@ Q2* RESULT Q!
 6     Q2* DUP 0< IF
 7       OP2 Q@ RESULT Q@  Q+  RESULT Q!
 8     THEN
 9   LOOP
10   2DROP 2DROP  RESULT Q@  ;
11
12
13
14
15


 0 ( QUADRUPLE WORD ARITHMETIC   Q* Q/ SUPPORT WORDS  )
 1
 2 : ?DSIGN-ABS    ( D#1 D#2 - |D#1| |D#2| BOOLEAN )
 3   >R >R >R I DABS R> 0<  R> SWAP I SWAP >R DABS R> R> 0<  XOR ;
 4
 5 : ?QMINUS   ( Q# BOOLEAN - Q# )
 6   IF
 7     >R >R >R >R  0 0 0 0  R> R> R> R>   Q-
 8   THEN   ;
 9
10 : ?QSIGN-ABS    ( Q#1 Q#2 - |Q#1| |Q#2| BOOLEAN )
11   >R >R >R >R >R  I 0<  R> SWAP DUP >R ?QMINUS R>
12   R> SWAP R> SWAP R> SWAP R> SWAP >R  DUP 0< DUP >R ?QMINUS
13   R> R>  XOR   ;
14
15                                   Listing continued on page 20
```

the operations check for overflow. Since most of the routines use temporary local variables to store intermediate results, most of the routines are not reentrant, but all of the routines are serially reusable. This means that in a multitasking environment, two tasks cannot use these routines at the same time, but two tasks can use the routines one after the other. This restriction was tolerated to avoid a complete dominance of the implementations by return stack manipulation.

Although these routines are not parameterized for quadruple word lengths, they are all obviously generalizable to "n" word length arithmetic. In some of these routines double word operators could significantly reduce complexity, size and execution time. They have been avoided in this implementation in favor of the obvious enhancement to larger word lengths.

I would like to thank Dr. Jon Louis Bentley of Carnegie-Mellon University for suggesting this particular division algorithm and Kim Harris for his aid in the multiplication algorithm.

*(Glossary on page 22)*

REFERENCES

Knuth, Donald E., *The Art of Computer Programming Volume 2 Seminumerical Algorithms*, Addison-Wesley Publishing Co., 1969, Chapter 4.3.1.

## Quadruple Word Simple Arithmetic (continued)

```
 0 ( QUADRUPLE WORD ARITHMETIC   Q* Q/  )
 1
 2 : Q/      ( Q#1 Q#2 - Q#RESULT )
 3   ?QSIGN-ABS >R  |Q/|
 4   >R >R >R >R 2DROP 2DROP R> R> R> R>  R> ?QMINUS   ;
 5
 6 : Q*      ( D#1 D#2 - Q#RESULT )
 7   ?DSIGN-ABS >R  |Q*|   R> ?QMINUS  ;
 8
 9
10
11
12
13
14
15
```

```
 0 ( QUADRUPLE WORD ARITHMETIC   D*/  )
 1
 2 : D*/      ( D#1 D#MULTIPLIER D#DIVISOR - D#RESULT )
 3   >R >R   Q*
 4   R> R> DUP 0< IF
 5     -1 -1
 6   ELSE
 7     0 0
 8   THEN
 9   Q/  2DROP  ;
10
11
12
13                  End of Listing
14
15
```

# THE FORTH SOURCE™

| *FORTH DISKS WITH DOCUMENTATION* | *PRICE* |
|---|---|
| **fig-FORTH Model and Source Listing,** with printed Installation Manual and Source Listing. ☐ APPLE II 5¼ ☐ 8080/Z80® 8 ☐ 8086/8088 8 ☐ H89/Z89 5¼ | $65.00 |
| ☐ **APPLE II/II+** by MicroMotion. Version 2. FORTH-79 Standard, editor, assembler, 200 pg manual, 5¼ | 100.00 |
| ☐ **APPLE II/II+ COMBO 1** by MicroMotion. Version 2. All of the above plus floating point and HiRes Turtle graphics | 140.00 |
| ☐ **APPLE II** by Kuntze. fig-FORTH editor, assembler, source listing and screens, 5¼ | 90.00 |
| ☐ **ATARI®** by Pink Noise Studio. fig-FORTH, editor, assembler, missile graphics, sound and handle drivers, 5¼ | 90.00 |
| ☐ **CP/M** by MicroMotion. Version 2.x. FORTH-79 Standard, editor, assembler. 200 pg manual, 8 | 100.00 |
| ☐ **CP/M Combo 1** by MicroMotion. 2.x. All of the above plus floating point. | 140.00 |
| ☐ **CROMEMCO®** by Inner Access fig-FORTH editor, assembler, 5¼ or 8 | 100.00 |
| ☐ **H89/Z89** by Haydon. fig-FORTH Stand Alone, source, editor, assembler & tutorial on disk. 5¼ | 250.00 |
| ☐ **H89/Z89** by Haydon. fig-FORTH, CP/M®, source, editor,assembler, & tutorial on disk, 5¼ | 175.00 |
| ☐ **HP-85** by Lange. fig-FORTH, editor and assembler, 5¼ | 90.00 |
| ☐ **IBM® PC/FORTH** by Laboratory Microsystems. fig-FORTH, editor and assembler. Manual, 5¼ | 100.00 |
| ☐ **IBM-Floating Point** by Laboratory Microsystems. Requires PC/FORTH. Specify software or for AMD 9511, AMD 9512 or Intel 8087 | 100.00 |
| ☐ **IBM-Cross Compiler** by Laboratory Microsystems. Requires PC/FORTH. (Nautilus Systems Model) | 300.00 |
| ☐ **PET®** by FSS. fig-FORTH editor and assembler, 5¼ | 90.00 |
| ☐ **PET®** with floating point, strings, disk I/O | 150.00 |
| ☐ **TRS-80/I** by Nautilus Systems. fig-FORTH, editor and assembler, 5¼ | 90.00 |
| ☐ **TRS-80/I or III** by Miller Microcomputer Services. MMSFORTH, FORTH-79 subset, editor, assembler, dbl-precision, arrays, utilities & applications. 210 pg. manual, 5¼ | 130.00 |
| ☐ **6800** by Talbot Microsystems. fig-FORTH, editor, assembler, disk I/O, FLEX® 5¼ or 8 | 100.00 |
| ☐ **6809** by Talbot Microsystems. fig-FORTH, editor, assembler, disk I/O, FLEX® 5¼ or 8 | 100.00 |
| ☐ **6809 Enhanced** 2nd screen editor, macroassembler,tutorial, tools and utilities, FLEX | 250.00 |
| ☐ **Z80** by Laboratory Microsystems. Editor and assembler, CP/M, 8 | 50.00 |
| ☐ Z80, floating point, requires Z80 above | 150.00 |
| ☐ Z80, AMD 9511 support, requires Z80 above | 150.00 |
| ☐ **Z80** by Inner Access. Editor, assembler and manual, CP/M, 8 | 100.00 |
| ☐ **8080** by Inner Access. Editor, assembler, and manual, CP/M, 8 | 100.00 |
| ☐ **8086/88** by Laboratory Microsystems. Editor, assembler, CP/M-86®, 8 | 100.00 |
| ☐ 8086/88 with floating point, CP/M-86 | 200.00 |
| ☐ 8086/88 with AMD 9511 support CP/M-86 | 200.00 |

| *FORTH PROGRAMMING DISKS* | *PRICE* |
|---|---|
| ☐ **"MVP-FORTH"** byHaydon & Boutelle. An extended program development system. Based on "All About FORTH" and optimized for CP/M and 8080/Z80. A ***public domain*** product. 8 inch | $ 75.00 |
| ☐ **"FORTH PROGRAMMING AIDS"** by Curry Assoc. Decompiler, Subroutine Decompiler, Callfinder and Translator requires fig-FORTH nucleus. Specify CP/M, 8" or Apple 3.3, 5¼ | 150.00 |

| *FORTH MANUALS, GUIDES, & DOCUMENTS* | |
|---|---|
| ☐ **"All About FORTH"** by Haydon. Ideograms (words) of fig-FORTH, FORTH-79, Starting FORTH and much more. A MUST! A ***public domain*** product. | $20.00 |
| ☐ **"FORTH Encyclopedia"** by Baker and Derick. A complete programmer's manual to fig-FORTH with FORTH-79 references. Flow Charted | 25.00 |
| ☐ **"Starting FORTH"** by Brodie. Prentice Hall. Best user's manual available. (soft cover) | 16.00 |
| ☐ **"Starting FORTH" (hard cover)** | 20.00 |
| ☐ **"METAFORTH"** by Cassady. Cross compiler with 8080 code. | 30.00 |
| Proceedings of Technical Conferences | |
| ☐ "1980 FORML" (FORTH Modification Laboratory) | 25.00 |
| ☐ "1981 FORML" Two Volume Set | 40.00 |
| ☐ "1981 Rochester University" | 25.00 |

| *MORE FORTH BOOKS & MANUALS* | | | |
|---|---|---|---|
| ☐ **"Systems Guide to fig-FORTH"** | 25.00 | ☐ **"APPLE® (MicroMotion) User's Manual"** | 20.00 |
| ☐ **"Using FORTH"** | 25.00 | ☐ **"CP/M® (MicroMotion) User's Manual"** | 20.00 |
| ☐ **"A FORTH Primer"** | 25.00 | ☐ **"TRS-80® MMSFORTH User's Manual"** | 18.50 |
| ☐ **"Caltech FORTH Manual"** | 12.00 | ☐ **"FORTH-79 Standard"** | 15.00 |
| ☐ **"Threaded Interpretive Languages"** | 20.00 | ☐ **"Tiny Pascal in fig-FORTH"** | 10.00 |
| ☐ **"Invitation to FORTH"** | 20.00 | ☐ **"FORTH-79 Standard Conversion"** | 10.00 |
| ☐ **"PDP-11 FORTH User's Manual"** | 20.00 | | |
| ☐ **"AIM FORTH User's Manual"** | 12.00 | | |

| *INSTALLATION DOCUMENTS* | |
|---|---|
| ☐ **Installation Manual for fig-FORTH,** contains FORTH model, glossery, memory map, and instructions **Source Listings of fig-FORTH,** for specific CPU's and computers. The above installation manual is required for implementation. Each | $15.00 15.00 |

☐ 1802 ☐ 6502 ☐ 6800 ☐ AlphaMicro
☐ 8080 ☐ 8086/88 ☐ 9900 ☐ APPLE II®
☐ PACE ☐ 6809 ☐ NOVA ☐ PDP-11/LSI/11

***CROSS COMPILER DISKS***

***Allows extending, modifying and compiling for speed and memory savings, can also produce ROMable code.***

***Nautilus (NS), Talbot Microsystems (TM), Laboratory Microsystems (LM) and Inner Access (IA).***

- ☐ *CP/M (NS) 200.00*
- ☐ *H89/Z89 (NS) 200.00*
- ☐ *TRS80/1 (NS) 200.00*
- ☐ *Northstar (NS) 200.00*
- ☐ *6809 (TM) 350.00*
- ☐ *IBM (LM)* 300.00*
- ☐ *8086 (LM)* 300.00*
- ☐ *Z80 (LM)* 200.00*
- ☐ *CP/M (IA) 450.00*
- ☐ *Cromemco (IA) 450.00*

** Requires FORTH disk*

**ORDERS ONLY (415) 961-4103** **DEALER & AUTHOR INQUIRIES INVITED**

**Ordering Information:** Check, Money Order (payable to MOUNTAIN VIEW PRESS, INC.), VISA or MasterCard accepted. No COD's or unpaid PO's. California residents add 6½% sales tax. Shipping costs in US included in price. Foreign orders, pay in US funds on US bank, include for handling and shipping by Air: $5.00 for each item under $25.00, $10.00 for each item between $25.00 and $99.00 and $20.00 for each item over $100.00. Minimum order $10.00. All prices and products subject to change or withdrawal without notice. Single system and/or single user license agreement required on some products.

**Specializing in the FORTH Language**

# MOUNTAIN VIEW PRESS, INC.

**PO BOX 4656** **MOUNTAIN VIEW, CA 94040** **(415) 961-4103**

(Continued from page 19)

# Quadruple Word Arithmetic Glossary

**1TEMP**
**2TEMP**
These quadruple word variables aid in the elimination of much of the return stack manipulation which otherwise would be required. They temporarily hold quadruple word numbers during the calculations of Q+TEMP and Q–TEMP. Their usage does prevent the algorithms from being reentrant.

**Q! ( Q# addr – )**
Stores a quadruple word number at the address given. The most significant word is stored at the address and the least significant word is stored at the address+6.

**Q@ ( addr – Q# )**
Fetches the quadruple word number from the address given. The most significant word ends up on the top of stack.

**?CARRY ( W#1 W#2 Result – boolean )**
Accepts the two inputs to an addition and the result of the addition and leaves a one if a carry occurred and a zero otherwise. It does this by comparing the signs of the inputs and the sign of the result using the following truth table.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| W#1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| W#2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| Result | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| boolean | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |

**?BORROW ( W#1 W#2 Result – boolean )**
Accepts the two inputs to a subtraction and the result of the subtraction and leaves a one if a borrow occurred and a zero otherwise. Because of the identity, if A–B = C then A = B + C, ?CARRY is used to compute the borrow.

**Q+TEMP ( – )**
The two inputs are passed in 1TEMP and 2TEMP. The result of the quadruple word addition is left in 2TEMP.

**Q–TEMP ( – )**
The two inputs are assumed to be in 1TEMP and 2TEMP. The result of the quadruple word stubtraction of 2TEMP from 1TEMP is left in 2TEMP.

**Q+ ( Q#1 Q#2 – Q#sum )**
Accepts two quadruple word operands on the stack and returns the result of their addition on the stack.

**Q– ( Q#1 Q#2 – Q#difference )**
Accepts two quadruple word operands on the stack and returns the result of the subtraction of Q#2 from Q#1 on the stack.

**Q2*TEMP ( – )**
The input is passed in 2TEMP. The result of it being mulitplied by two is left in 2TEMP.

**Q2/TEMP ( – )**
The input is passed in 2TEMP. The result of it being divided by two is left in 2TEMP.

**Q2* ( |Q#| – |Q#| )**
Accepts a quadruple word number on the stack and multiplies it by two. Note that since the number is an absolute value the high order bit will always be zero thus preventing overflow from occurring.

**Q2/ ( |Q#| – |Q#| )**
Accepts a quadruple word number on the stack and divides it by two. Note that since the number is an absolute value the high order bit can always be made a zero without consideration of a possible sign bit on the number.

**Q< ( Q#1 Q#2 – boolean )**
Compares the two quadruple word operands and leaves a one if Q#1 is less than Q#2 and a zero otherwise.

**Q0> ( Q#1 – boolean )**
Compares the quadruple word number on the stack with zero and leaves a one if it is greater than zero and a zero if the number is less than or equal to zero.

**DIVISOR**
**REMAINDER**
**QUOTIENT**
**POWER**
These quadruple word variables are used to store temporary intermediate results during Q/. These variables aid in the elimination of much of the return stack manipulation which otherwise would be required. Their usage prevents the algorithms from being reentrant.

**RESULT**
**OP2**
These quadruple word variables are used to store temporary intermediate results during Q*. These variables aid in the elimination of much of the return stack manipulation which otherwise would be required. Their usage prevents the algorithms from being reentrant.

**@POWER ( – )**
The inputs are passed in DIVISOR, REMAINDER and POWER. This word multiplies the divisor by two until it exceeds the dividend. It then backs up one multiplication.

**@QUOTIENT ( – )**
The inputs are passed in DIVISOR, REMAINDER, QUOTIENT and POWER. This word repetitively subtracts all lower multiples of two of the DIVISOR accumulating a quotient and obtaining a remainder.

**|Q/| ( |Q.dividend| |Q.divisor| – |Q.remainder| |Q.quotient| )**
Using the absolute value of the dividend and the divisor, the absolute value of quotient and remainder are computed.

**|Q*| ( |D#1| |D#2| – |Q.result| )**
The absolute values of the two double word operands are multiplied giving an absolute value quadruple word result. The algorithm merely looks at each bit of the first operand and for every one bit adds an appropriately shifted second operand to the accumulating result. This algorithm could be speeded up in many ways. The first is by looking at the counter operand two bits at a time (suggested by Kim Harris). A second is by initially examining the operands for the one with the least number of ones to use as the counter. A third way is to examine the remaining counter at each iteration for a zero value and allowing the loop to terminate early.

**?DSIGN-ABS ( D#1 D#2 – |D#1| |D#2| boolean )**
This word computes the absolute value of the two double word numbers input on the stack and in addition to them leaves a one if the two numbers were of differing sign or a zero if the numbers were of the same sign. This word is used to calculate the sign of the result of multiplication as well as taking the absolute values of the operands which the multiply algorithm requires.

**?QMINUS ( Q# boolean – Q.result )**
If the boolean on the top of the stack is a one, the quadruple word number is two's complemented. This word is used to correct the sign of the result of a multiply or divide.

**?QSIGN-ABS ( Q#1 Q#2 – |Q#1| |Q#2| boolean )**
This word computes the absolute value of the two quadruple word numbers input on the stack and in addition to them leaves a one if the two numbers were of differing sign or a zero if the numbers were of the same sign. This word is used to calculate the sign of the result of division as well as taking the absolute values of the operands which the divide algorithm requires.

**Q/ ( Q#dividend Q#divisor – Q#quotient )**
The quadruple word quotient resulting from the division of Q#1 by Q#2 is left on the stack.

**Q* ( D#1 D#2 – Q#result )**
The quadruple word result of the multiplication of Q#1 and Q#2 is left on the stack.

**D*/ ( D#1 D#multiplier D#divisor – D#result )**
This is the typical */ but with double word inputs and outputs. A full quadruple word intermediate result is used.

# Floating Point FORTH?

By MICHAEL JESCH
Aregon Systems, Inc.

One of the first things most programmers find missing in FORTH is floating point arithmetic. While most implementers of FORTH probably weigh the advantages and adversities of floating point for their version, they usually decide to forego it for various reasons. On the other hand, some excellent floating point systems have been developed in FORTH. This article overviews some major problems with floating point numbers, and examines a very rudimentary floating point system, written entirely in high level FORTH.

The first major problem with floating point numbers is that no computer can work with numbers that are truly floating point. Instead, quite often they are stored as two separate numbers, mantissa and exponent. This leads to

**The greatest advantage of a floating point math system comes in ease of use**

the second major problem, that of speed. Because each floating point number is stored as two separate numbers, each function requires that two numbers be dealt with, costing quite a bit in speed. One of the most common solutions for speed problems is to buy a dedicated processor chip to do all the arithmetic. This is impossible on some computers, and costly on others.

Another problem, that of accuracy, is a prime consideration. While floating point numbers can have a greater range, their precision suffers a little. The system outlined in this article has precision to only six full decimal digits, but the decimal point can be moved 127 places in either direction. This compared to a normal 32 bit double length number, where the range and accuracy are +/−2,147,483,647.

The greatest advantage of a floating point math system comes in ease of use: there is no need for the programmer to worry about where the decimal point should be, as it is handled internally by the floating point operators themselves. This saves time programming and debugging, and usually saves some memory too.

*Continued on next page*

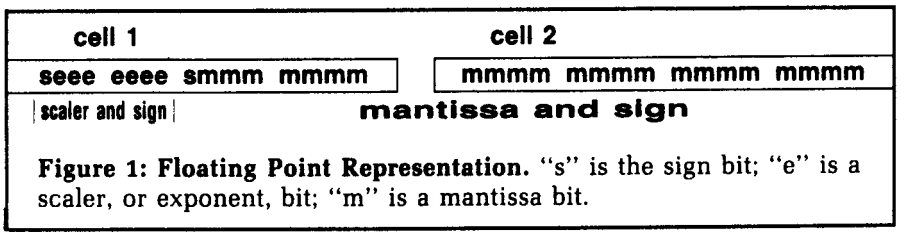



**Figure 1: Floating Point Representation.** "s" is the sign bit; "e" is a scaler, or exponent, bit; "m" is a mantissa bit.

```
1000  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1
  2  VARIABLE  FPSW          ( floating point status word )
  3
  4  FPSW 1+ CONSTANT FBASE      ( base )
  5
  6  : FRESET   ( -- )      ( clear condition codes )
  7       0 FPSW C! ;
  8
  9  : FINIT    ( -- )      ( initialize processor )
 10       FRESET
 11       BASE @   FBASE ! ;
 12
 13
 14
 15


1001  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1
  2  : FER   ( -- n )   ( returns sum of condition codes )
  3       FPSW C@ ;
  4
  5  : FZE   ( -- n )   ( true if last F# was zero )
  6       FER  1 AND  0= NOT ;
  7
  8  : FNE   ( -- n )   ( true if last F# was < zero )
  9       FER  2 AND  0= NOT ;
 10
 11  : FOV   ( -- n )   ( true if last operation was overflow )
 12       FER  4 AND  0= NOT ;
 13
 14  ( CC's 8, 16, 32, 64, and 128 are available )
 15


1002  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1  HEX
  2
  3  : SFZ   ( F# -- F# ; Z )   ( sets Z according to F# )
  4       FER  FFFE AND  FPSW C!     ( reset Z )
  5       2DUP  00FF AND  D0=   FER OR   FPSW C! ;
  6
  7  : SFN   ( F# -- F# ; N )   ( sets N according to F# )
  8       FER  FFFD AND  FPSW C!     ( reset N )
  9       DUP  0080 AND  40 /  FER OR  FPSW C! ;
 10
 11  DECIMAL
 12
 13
 14                                  Listing continued on next page
 15
```

This floating point system is written in high level FORTH as an educational tool. Once the machine language of the target machine is understood, it should be rewritten in low level to capitalize on speed. One important side effect/advantage of this approach is transportability: It has since been implemented on two other computers, under different FORTH systems (polyFORTH and MMSFORTH); one with a different CPU (an LSI-11/23). This approach does, however, cost a lot of execution time.

As can be seen in Figure One, the floating point number is represented in two 16-bit cells on the stack. The high cell (cell 1) contains the 8-bit exponent (containing one sign bit) and the high 8 bits of the mantissa, including the mantissa sign, while the low cell (cell 2) contains the lower 16 bits of the mantissa. In this manner, the existing double length stack and memory operators can be used to manipulate the values.

The error detection is handled by the system, but the recovery is left up to the programmer. A special condition code 'register' is used to return information about the last operation. Currently, three of these bits are used: One each to indicate the occurrence of a zero value, a negative value, and most overflow/underflow conditions. These flags can be tested, as in a FORTH **IF . . . THEN** structure, with words defined in the package (**FZE FNE** and **FOV**). The word **FER** will return a true if any error existed.

The scaler is used to tell where the decimal point is, relative to the ones' column of the mantissa, during all math operations and output. A positive value indicates that the radix point is actually to the right of the ones' column and by how many digits, while a negative value means to move it to the left. It could be considered a 'times **BASE** to the **SCALER**' type of suffix to a number. For addition and subtraction, the scalers must be made equal (the word **ALIGN** does this). This means shifting the mantissa the number of places equal to the difference of the exponents, which

```
1003  LIST

 0  ( Floating point      MCJ   11/02/81 )
 1  HEX
 2
 3  : @EXPONENT   ( F# -- M E ;  Z N )   ( remove exponent )
 4       FRESET   SFZ   SFN              ( set flags )
 5       DUP  FF00 AND  100 /  >R        ( obtain exponent )
 6       FNE IF                          ( sign extend mantissa )
 7         FF00 OR
 8       ELSE
 9         00FF AND
10       THEN  R> ;
11
12  DECIMAL
13
14
15


1004  LIST

 0  ( Floating point      MCJ   11/02/81 )
 1  HEX
 2
 3  : !EXPONENT   ( M E -- F# ;  V Z N )   ( restores exponent )
 4       DUP  100 *   DUP 100 /   ROT <> IF
 5         4  FPSW C!                        ( exponent overflow )
 6       THEN
 7       SWAP DUP  FF00 AND DUP IF
 8         DUP  FF00 <> IF
 9           4 FPSW C!              ( mantissa overflow )
10         THEN
11       THEN DROP
12       00FF AND   OR
13       SFZ SFN ;
14
15  DECIMAL


1005  LIST

 0  ( Floating point      MCJ   11/02/81 )
 1  : F.   ( F# -- ;  Z N )
 2       @EXPONENT >R
 3       SWAP OVER  DABS
 4       <#   I 0< IF
 5         I ABS  0  DO   #   LOOP        46 HOLD
 6       ELSE
 7         46 HOLD
 8         I IF
 9           I  0  DO   48 HOLD   LOOP
10         THEN
11       THEN   R> DROP
12       #S   SIGN #>  TYPE SPACE ;
13
14  : E.   ( F# -- ;  Z N )
15     @EXPONENT  <ROT  (D.) TYPE  ." . E" . ;


1006  LIST

 0  ( Floating point      MCJ   11/02/81 )
 1
 2  : F*   ( F#1 F#2 -- F# ; N Z V )   ( multiply )
 3       2SWAP  @EXPONENT >R
 4       2SWAP  @EXPONENT >R
 5       DROP  1  M*/
 6       R>  R> +   !EXPONENT ;
 7
 8
 9  : F/   ( F#1 F#2 -- F# ; N Z V )   ( multiply )
10       2SWAP  @EXPONENT >R
11       2SWAP  @EXPONENT >R
12       DROP  1 SWAP  M*/
13       R>  R> +   !EXPONENT ;
14
15
```

```
1007  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1
  2  : ALIGN   ( M1 E1  M2 E2 -- M1 M2 E )
  3       BEGIN
  4         >R  ROT >R
  5       I' I <> WHILE
  6         I' I > IF     ( I' is E2 )
  7           2SWAP   FBASE C@  1  M*/   2SWAP   R> 1+ <ROT  R>
  8         ELSE
  9           R> <ROT   FBASE C@  1 M*/  R> 1+
 10         THEN
 11       REPEAT
 12       R>   R> DROP ;
 13
 14
 15


1008  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1
  2  : F+   ( F#1  F#2  -- FSUM  ; N V Z )
  3       2SWAP  @EXPONENT >R
  4       2SWAP  R> <ROT  @EXPONENT
  5       ALIGN   >R
  6       D+   R> !EXPONENT ;
  7
  8  : F-   ( F#1  F#2  -- FDIFF  ; N V Z )
  9       2SWAP  @EXPONENT >R
 10       2SWAP  R> <ROT  @EXPONENT
 11       ALIGN   >R
 12       D-   R> !EXPONENT ;
 13
 14
 15


1009  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1
  2  : RSCALE   ( F# -- F# ; N Z V )
  3       @EXPONENT 1+  <ROT
  4       FBASE C@  1 M*/  ROT
  5       !EXPONENT ;
  6
  7  : LSCALE   ( F# -- F# ; N Z V )
  8       @EXPONENT 1-  <ROT
  9       1  FBASE C@ M*/  ROT
 10       !EXPONENT ;
 11
 12
 13
 14
 15


1010  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1  : FIX   ( F# -- D# ; V Z N )
  2       @EXPONENT
  3       BEGIN
  4       ?DUP  WHILE
  5         DUP 0< IF
  6           1+  <ROT   FBASE C@  1  M*/
  7         ELSE
  8           1-  <ROT   1  FBASE C@  M*/
  9           2DUP D0= IF
 10             5 FPSW C!  ROT DROP 0 <ROT   ( underflow )
 11           THEN
 12         THEN
 13         ROT
 14       REPEAT ;
 15
```

*Listing continued on next page*

causes most of the imprecision problems present in this system. Furthermore, if one number was entered in hexadecimal (base 16) and the other in decimal, the scalers would be incompatible. To help circumvent this, a floating point base value is kept separate from the FORTH base value, and all internal scaling oprations use this value for the number base. The floating point base is set to the current FORTH base when the floating point system is initialized (with **FINIT**). It can also be explicitly set by the programmer, but be careful with this; if you output a number in a different base than you did arithmetic, the results will not be correct.

Number formatting is left up to the programmer, as it is in most FORTH systems. A double length number may be converted to floating point by inserting the desired scaler (number scaler **!EXPONENT**). To change a number from floating point to integer, the word **FIX** will scale the mantissa to zero. The scaler of the top floating point number can be extracted with the word **@EXPONENT**, which leaves the double precision mantissa below, unmodified. **F.** and **E.** are used to output the top floating point number. **F.** prints in floating point format (i.e., 123.45), while **E.** prints in scientific notation (i.e., 12345 E −2).

The four basic arithmetic functions, add, subtract, multiply and divide, are called **F+**, **F−**, **F*** and **F/**, respectively. **RSCALE** and **LSCALE** are used to change the position of the least significant digit in the mantissa. **RSCALE** decrements the scaler and multiplies the mantissa by base (changes 12.3 to 12.30), while **LSCALE** increments the scaler and divides the mantissa by base (changes 12.34 to 12.3). Be careful with these words, however. If the number is close to the limit of precision, the number will probably lose accuracy.

Other miscellaneous words are **FABS**, **FNEGATE**, **FMIN**, **F>**, and **FMAX**; these are the floating point counterparts to **ABS**, **NEGATE**, **MIN**, **>**, and **MAX**, respectively.

# Vendor Support of Floating Point

Not everyone feels the same about the need for floating point arithmetic in a FORTH system. This variance of opinion is reflected in FORTH products. Some vendors, such as FORTH, Inc., generally don't support floating point at all, for philosophical reasons.

Full floating point FORTH systems, designed to support floating point processors, are available from other vendors. While these systems can be fast, they are usually expensive.

Other vendors have written complete floating point packages from scratch, with low-level arithmetic functions written in assembler for speed, and the higher-lever FORTH words built upon them. MicroMotion is one vendor that has taken this approach.

Yet another way to provide floating point arithmetic is by supporting calls to ROM-resident subroutines for the particular micro. This is the method used by MMSFORTH's Floating Point Utility, for example.

Here are descriptions of the two floating point packages mentioned earlier, as representative of what's currently available on the market:

## MMSFORTH Floating Point Utility

MMSFORTH offers an optional floating point package that uses the TRS-80's ROM-resident subroutines.

The expected mathematical primitives: add, subtract, multiply, divide, change sign, and literal operators are present for the 24-bit mantissa, 8-bit exponent single-precision numbers, the 56-bit mantissa, 8-bit exponent double-precision numbers, and the complex numbers (consisting of a single-precision real part, and a single-precision imaginary part).

Single precision arithmetic includes log, trig and floating point random number functions. Complex numbers are enhanced with the magnitude, phase (radians or degrees), conjugate, rectangular-to-polar, and polar-to-rectangular functions.

MMSFORTH user Jim Gerow reports: "As far as the performance aspects of the floating point package, it is just a little faster than Basic when it comes to solving floating point equations. When compared with FORTH's integer and double-integer (i.e., fixed-point) operations, it is no contest — the extra housekeeping that the floating point operations have to perform really slow it down. Nevertheless, the floating point package has allowed me to solve some pretty detailed engineering equations. I've found the MMSFORTH package to be very complete and accurate."

## MicroMotion FORTH-79 Floating Point Arithmetic Package

MicroMotion offers what they call "the most comprehensive Forth floating point arithmetic package on the market, fifty functions in all." According to MicroMotion's Phil Wasson, the Floating Point Package consists of an 11 page manual and disk. The system is available for Apple II, CP/M, NorthStar DOS, and Cromemco CDOS, in a wide variety of disk formats. The price is $49 (the FORTH-79 system is $99).

The manual includes a glossary summarizing each of the fifty functions.

Besides the usual four functions, words are included to convert between strings, single-numbers, double-numbers and floating point numbers. Twenty-four trigonometric and hyperbolic functions are also included.

The low-level arithmetic functions are written in assembler (6502 and Z-80) for speed, with the higher-level words built upon them. Output conversion works in any BASE. Source code is provided.

The floating point format is that used by the AMD 9511 and the Intel 8231. The internal format is 32-bits: 24-bit mantissa, 6-bit exponent, and a sign bit for each. Numbers can range from (decimal) plus or minus 2.7 times 10 to the −20th power, up to 9.2 times 10 to the 18th power. Numbers have seven significant digits of precision.

### Floating Point FORTH? (continued)

```
1011  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1
  2  : FABS    ( F# -- |F#| ;  N Z V )
  3       @EXPONENT  <ROT  DABS  ROT  !EXPONENT ;
  4
  5  : FNEGATE    ( F# --  -F# ; N Z V )
  6       @EXPONENT <ROT   DNEGATE  ROT  !EXPONENT ;
  7
  8  : FMIN    ( F#1 F#2 -- m[F#s] ; N Z V )
  9       2SWAP  @EXPONENT >R
 10       2SWAP R> <ROT  @EXPONENT
 11       ALIGN >R
 12       DMIN  R> !EXPONENT ;
 13
 14
 15

1012  LIST

  0  ( Floating point      MCJ   11/02/81 )
  1
  2  : F=    ( F#1 F#2 -- b ;  N Z V )
  3       F-  2DROP  FNE ;
  4
  5  : FMAX    ( F#1 F#2 -- M[F#] ; N Z V )
  6       2OVER  2OVER  FMIN  F- F+ ;
  7
  8
  9
 10
 11
 12
 13
```

# FPA FORTH PROGRAMMING AIDS by Curry Associates

**FORTH PROGRAMMING AIDS** is a software package containing high level FORTH routines which allow the FORTH programmer to write more efficient programs more quickly, and they are especially useful for those who are using a metacompiler or cross compiler.
FORTH PROGRAMMING AIDS allow the programmer to

- Minimize memory requirements for target systems by finding only those words used in the target application.
- Tailor existing words (including nucleus words) to specific needs by decompiling the word to disk, editing, and recompiling.
- Build on previous work by transferring debugged FORTH routines (including constants and variables) from one application to another.
- Speed program loops by finding **all** words called from the loop for possible merging or recoding to assembler.
- Patch changes into compiled words in seconds.

---

**FORTH PROGRAMMING AIDS** comes with complete source code and a 40-page manual. FPA contains the following modules:

---

**DECOMPILER** This is a **true** decompiler which converts the FORTH words in RAM into compilable, structured FORTH source code, including program control words such as IF, ELSE, THEN, BEGIN, etc. If you ask FPA to DECOMPILE the nucleus word INTERPRET, you get the following output displayed on your terminal within 3 seconds:

```
( NFA&PFA: 4094 4108 )
  : INTERPRET
       BEGIN -FIND
          IF STATE @ <
             IF CFA ,
             ELSE CFA EXECUTE
             THEN ?STACK
          ELSE HERE NUMBER DPL @ 1+
             IF [COMPILE] DLITERAL
             ELSE DROP [COMPILE] LITERAL
             THEN ?STACK
          THEN
       AGAIN ;
```

You may DECOMPILE one word, or a range of words at one time — even the whole FORTH system! This decompliled output may be sent by FPA options to the console, printer, or to disk. DECOMPILE is useful for looking up words, or for obtaining variations of words by decompiling to disk, editing, and recompiling.

**SUBROUTINE DECOMPILER** The subroutine decompiler finds words called by a specified word to all nesting levels. This makes FORTH PROGRAMMING AIDS especially useful for metacompilers or cross compilers and for finding words called within a loop. The found words may be DECOMPILEd to disk.

**CALLFINDER** This set of routines is designed to find the calls to a specified word or set of words, either within the context vocabulary or across all vocabularies. Useful to locate and merge infrequently called words, or to see which words will be affected by changing the specified word.

**TRANSLATOR** This program provides a one-to-one translation of the high level FORTH words in RAM. (This is sometimes called decompilation, but the output is not suitable input for a FORTH compiler). Useful for debugging, patching into compiled words, etc.

**System Requirements** FORTH nucleus based on the fig-FORTH model or 79-STANDARD; a minimum of 3K bytes and a recommended 13K bytes of free dictionary space. FORTH PROGRAMMING AIDS can be used on itself to generate minimum size modules.

---

Yes, send me a copy of FORTH PROGRAMMING AIDS, including all source code and the 40-page manual.

| | | |
|---|---|---|
| ☐ fig-FORTH model | $150 | Calif. residents add 6.5% tax. |
| ☐ FORTH-79 STANDARD (specify system) | $150 | Foreign air shipments add $15. |
| ☐ Manual alone (credit toward program purchase) | $25 | |
| ☐ Send more information | | |

☐ Master Charge ☐ Visa Account Number _____ Exp. Date _____

Name _____

Company _____

Street _____

City _____ State _____ Zip _____

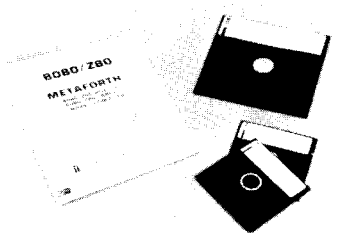
Indicate disk format:

- ☐ 8" ss/sd fig-FORTH screens
- ☐ 8" ss/sd CP/M™ 2.2 file
- ☐ Apple 3.3
- ☐ PC FORTH
- ☐ Other _____

Send to: **Curry Associates • P.O. Box 11324 • Palo Alto, CA 94306**

# DEVELOPMENT TOOLS

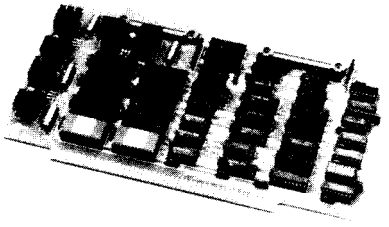
**Develop FORTH code for any target 8080/Z80 system on your current 8080/Z80 or Cromemco CDOS based system**




### 8080/Z80 METAFORTH CROSS-COMPILER

- Produces code that may be downloaded to any Z80 or 8080 processor
- Includes 8080 and Z80 assemblers
- Can produce code without headers and link words for up to 30% space savings
- Can produce ROMable code
- 79 Standard FORTH
- Price $450

**No downloading – No trial PROM burning. This port-addressed RAM on your S-100 host is the ROM of your target system**



### WORD/BYTE WIDE ROM SIMULATOR

- Simulates 16K bytes of memory (8K bytes for 2708 and 2758)
- Simulates 2708, 2758, 2516, 2716, 2532, 2732, 2564 and 2764 PROMS
- The simulated memory may be either byte or 16-bit word organized
- No S-100 memory is needed to hold ROM data
- Driver program verifies simulated PROM contents
- Price $495 each

## CONSULTING SERVICES

Inner Access provides you with Custom Software Design. We have supplied many clients with both Systems and Application Software tailored to their specific needs. Contact us for your special programming requirements.

## FORTH WORKSHOPS

**ONE-WEEK WORKSHOPS — ENROLLMENT LIMITED TO 8 STUDENTS**

### FORTH Fundamentals

- Program Design
- Program Documentation
- FORTH Architecture
- FORTH Arithmetic
- Control Structures
- Input/Output
- The Vocabulary Mechanism
- Meta-Defining Words

**JUNE 7-11 JULY 12-16**
**AUG. 9-13 SEPT. 13-17**

$395 Incl. Text

### Advanced FORTH Applications

- FORTH Tools
- Engineering Applications
- Floating Point
- Communications
- Sorting & Searching
- Project Accounting System

**JUNE 14-18**
**AUG. 16-20**

$495 Incl. Text

### Advanced FORTH Systems

- FORTH Internals
- Assemblers and Editors
- Other Compilers
- Cross-Compilation Theory
- Romability, Multitasking, Timesharing
- File Systems/ Database Systems

**JULY 19-23**
**SEPT. 20-24**

$495 Incl. Text

Instructors: LEO BRODIE, GARY FEIERBACH and PAUL THOMAS

(For further information, please send for our complete FORTH Workshop Catalog.)

## Inner Access Corporation

P.O. BOX 888 • BELMONT, CALIFORNIA 94002 • (415) 591-8295

BANKAMERICARD VISA

master charge

# FORTH—83 Preview

By JOHN S. JAMES
Colon Systems

The FORTH Standards Team met May 11 - 15, 1982 near Washington, D.C. to decide on changes to the FORTH-79 Standard. Eleven voting members of the team were present, and a two-thirds vote was required to alter the standard. The result of the meeting will be published as the FORTH-83 Proposed Standard, Draft 1; and a meeting tentatively scheduled for October 1982 will consider further changes. This article, based on the author's personal notes and recollection, is an unofficial summary of the changes from the point of view of the user; those which affect only the system implementer are not covered.

In brief, the meeting was highly successful. We went into it with the FORTH-79 standard, which is good in most respects but has a handful of serious deficiencies, as well as numerous minor errors in the published standards document. We came out with a more flexible and efficient language. Only a few of the changes will affect existing programs, and those changes were made for good reasons.

### Vocabularies

The most vehement criticism of FORTH-79 concerns vocabularies. All of them had to chain to FORTH, so sealed vocabularies and multi-level trees were apparently forbidden. FORTH-79 clarifies the wording to permit arbitrary vocabulary structures in a standard system, but the default search order is **CONTEXT**, then **FORTH**.

Strictly speaking, a standard program has no more freedom than before, because any vocabulary structure outside of the default is unspecified and therefore nonstandard. (And vocabulary usage in FORTH-79 programs will not need to be changed to run under FORTH-83.) The proposed FORTH-83, however, allows experimentation with more powerful structures, such as vocabulary stacks, as nonstandard structures within standard systems. There was a widespread feeling in the team that we do not understand vocabularies well enough to specify an adequate standard completely. In the future, one or more of the experimental structures, which are allowed by the new standard, will probably become predominant and made official.

### Loops

**LOOP** and **+LOOP** in FORTH-79 are inefficient, and prone to error because of irregularities concerning data structures which cross the 32K boundary. Two desirable alternatives were considered at the meeting, and the most efficient of them was accepted. The new loop has a range of 64K, and terminates when the index crosses the boundary between 'limit' and 'limit' minus one. Except in unusual cases such as **-10 0 DO...LOOP**, it behaves as the current loop does, for signed or unsigned loop limits. There is no singularity at the 32K address point. **LEAVE** leaves immediately, as it probably should, although the change will require modification of some FORTH-79 programs if they are converted to FORTH-83.

### Mono-addressing

Another important change is mono-addressing. In FORTH-79, some words use the pfa (parameter-field address), and others use cfa (code-field address) to refer to dictionary entries. Having both addresses causes unnecessary complication. The proposed FORTH-83 draft requires a single address, which after much controversy was specified as the pfa. (The cfa was also considered and it is possible that the final standard will specify only the behavior of the address and leave the actual choice to the implementer; in this case, another means of addressing into **DOES>** words would be required.)

**FIND**, while being changed to return the pfa instead of the cfa, was also changed to return "true" if found and "false" if not. **EXECUTE** was changed to accept the pfa. A rule of usage was added to prevent the pfa from being used to store into constants. Some modification of existing programs, especially those using advanced or tricky coding techniques, will be necessary.

### TIC, etc.

The remaining change which in this author's opinion is a major one is the elimination of "state-smart" words from the standard. The words affected are "tic", **LITERAL**, **."**, and new words "bracket-tic" and **.(**. (Note—the punctuation convention used in this article to distinguish FORTH from English words is to boldface each FORTH word, unless the word contains the single-quote character. In that case, the standard pronunciation is spelled out, and enclosed in double quotes.)

The major consequence of this change is that "tic" now conforms to the FORTH, Inc. usage (which is described in Starting FORTH); it is compiled as a regular word, and later looks ahead in the input stream when the new word which contains the "tic" is executed. In FORTH-79, as in the FIG model, "tic" inside a colon definition would look ahead immediately and compile the address of the next word in the input stream as a literal (the "smart tic"). This function inside a colon definition will now be handled by "bracket tic" in the FORTH-83 draft. There is much history behind this issue, but support for the change was so strong that the "smart tic" is probably gone for good.

In a related change, **."** can no longer be used outside of a colon definition (otherwise, it would have been the only state-smart word required in the standard). A word **.(** was added as a substitute for those uncommon cases, such as load screens or some menu techniques, where the function might be wanted. It types the following text immediately, and may be used inside or outside a colon definition.

Although all state-smart words have been eliminated from the standard, **STATE** itself is still included so that standard programs can define such words if they want to.

### Block I/O

One other change is likely to affect existing programs: block I/O. **BUFFER**, seldom used and full of hidden problems, was removed from the required standard (and put into the reference word set). The name **SAVE-BUFFERS** was changed to **FLUSH.** And programs are no longer allowed to put garbage into block buffers on the assumption that they will never be written to disk if they are not **UPDATE**'d; this restriction allows efficient use of disks with physical track sizes greater that 1024 bytes.

*(Continued next page)*

# NEW PRODUCT ANNOUNCEMENTS

## COLORFORTH for TRS-80 Color Computer

10K of 6809 FORTH in TRS-80 Color Computer ROM PAC. Runs on 4K, 16K, 32K or 64K models. Consists of Fig FORTH with most of FORTH-79 a la *Starting FORTH* book. Mass storage via cassette interface. Words to interface to high-res color graphics, joy sticks, sound.

Enhanced *Starting FORTH* editor included in ROM PAC. Split screen display. Contains dissassembler and other debugging aids. Assembler and execution simulator available on cassette.

112-page manual describes hardware features of the implementation, including a glossary and full source code listing. Manual available separately for $20; credit toward later product purchase.

Price: $109.95, includes manual and shipping. CA residents add 6% sales tax. Hardware warranted.

**The Micro Works • P.O. Box 1110 • Del Mar, CA 92014**
**Contact Ray Talbot, 213/376-9941 eves.**

---

## FORTH FOR PET

Fig-FORTH adapted for the Commodore/PET and the CBM Disk Drives, on one 5" floppy. Each block is stored in four consecutive disk sectors, beginning at track 1. Access is by track and sector number. Capacity is 150 screens on the 2040 or 4040 CMB drive; 480 screens on the 8050. The FORTH program, or an individualized FORTH, may be saved to disk.

Includes line-editing features of the Fig-FORTH model editor, screen editor, a full FORTH assembler, optional screens to conform to 79-Standard, printer interface, string and array functions. PET file I/O utilities, and clock and calendar functions.

A 79-page beginner's tutorial on how to use FORTH FOR PET is included, or may be purchased separately for $10, creditable to later purchase of system.

Price: $50. Shipping adds $1.25 to prepaid orders.

**A B Computers Inc. • 252 Bethlehem Pike • Colmar PA 18915 • 215/822-7727**
**Contact Gene Beals**

---

## OmniFORTH for TRS-80 Model III*

OmniFORTH, based on Fig-FORTH and the 79-Standard, contains the interactive OmniFORTH compiler, Z80 assembler, file system and full screen video editor. Requires 32K memory and one disk drive.

The package comes on disk complete with user's manual for $130 postpaid. Florida residents add sales tax. Foreign orders include additional postage and U.S. funds.

**Interactive Computer Systems, Inc. • 6403 DiMarco Road • Tampa, FL 33614**
**813/884-5270**

**TRS-80 is a trademark of the Tandy Corporation.*

---

## proFORTH

proFORTH is designed specifically for developing dedicated and real-time applications. Software development is done on the "target processor emulator" in the host development system. When testing is completed a ROM image of the target code memory is installed in the target hardware for execution.

Multiple target ROM, target RAM and dictionary (system) memory areas allow interactive symbolic testing throughout the development cycle.

With few exceptions, proFORTH is a compatible superset of the FORTH-79 Standard, including a unique name compression algorithm (3-characters, length, and hash code in 4 bytes); purgable dictionaries to conserve memory in applications over 32K; positional, keyed and conditional case statements; and a forward reference capability.

The 498-page manual is available separately for $100 (credited towards purchase).

The host system is the Tektronix Microprocessor Development Lab Models 8002 and 8550 with minimum 32K. Target processors are the 8080, 8085 and Z80.

Price: $2250 plus tax, including manual, shipping, six-month limited warranty and four hours telephone consultation.

**Microsystems, Inc. • 2500 E. Foothill Blvd., Suite 102 • Pasadena, CA 91107**
**213/577-1471 • Contact Robert H. Hertel**

---

## OGI Fig-FORTH

OGI Fig-FORTH is an inexpensive version of Fig-FORTH for the Apple II and Apple II+ computers with DOS 3.3. It includes sample screens, 6502 assembler, editor, decompiler, and APPLE cursor and printer control, on a 5" diskette. The 50-page manual includes release notes, installation guide, and glossary, and is available separately for $20, creditable toward purchase of a system.

Price: $40, includes U.S. postage.

**On-Going Ideas • RD #1, Box 810 • Starksboro, VT 05487 • 802/453-4442**
***Contact Hal Clark***

---

## DISK*FIX

DISK*FIX is a disk editor for Marx FORTH for use with Northstar single or double density disk systems. It allows complete editing of any byte of any sector displayed in either ASCII or hex, to rebuild crashed disks, etc. Includes automatic rewrite and verify modes that track down and fix problem sectors. Density switching allows editing of mixed density diskettes.

Also includes utilities to sort directories, rename files, and catalogue disks on a printer. Requires Z-80 and addressable cursor functions.

DISK*FIX is sold separately for $75; with Mark FORTH $130.

**Perkel Software Systems • 1636 N. Sherman • Springfield, MO 65803 • 417/862-9830**
**Marc Perkel**

### Byte moves

**CMOVE**, etc. were greatly improved, but in a way which will not normally affect existing programs. A backwards **CMOVE**, named **CMOVE>** and pronounced "cmove-up", will be a standard word. The confusing **MOVE** of FORTH-79 was eliminated, and a new **MOVE**, which is smart enough to transfer bytes in all cases and avoid propagation, was added to the reference word set only. **FILL** and the move words now take unsigned arguments.

### Other changes

In other changes, the definition of **WORD** was clarified, and changed so that it returns a blank, not the actual delimiter found, at the end of the word—the traditional FORTH behavior before FORTH-79. **WORD** will get more attention at the October meeting. The word **ABORT"** is required, for easy error halts. **2/** was also added to the Standard. And many typos, consistency errors, and unspecified conditions such as math errors, were fixed.

A change important for the future is the addition of a "system extension word set". Now words can be provided for the system programmer without being required in all standard systems shipped. A set of control primitives, which allow users to create new control structures comparable to **IF** and **WHILE**, is included in the new category.

There is also an experimental proposal section, but the May meeting did not have time to deal with experimental proposals except in a few cases.

### How to Submit Proposals

The draft of the proposed FORTH-83 standard will be published as soon as possible. New proposals, which can be considered at the October meeting, or possibly later by mail vote of the team, should be submitted on forms available from John Bumgarner, FORTH Standards Team, P.O. Box 4545, Mountain View, CA 94040. Your proposal can be distributed to team members prior to the October meeting if it is received by September 15, 1982.

# Fig Chapters

## U.S.

### Arizona

**Phoenix Chapter**
Peter Bates at 602/996-8398

### California

**Los Angeles Chapter**
Monthly, 4th Sat., 11 a.m., Allstate Savings, 8800 So. Sepulveda Blvd., L.A. Philip Wasson 213/649-1428

**Northern California Chapter**
Monthly, 4th Sat., 1 p.m., FORML Workshop at 10 a.m. Palo Alto area. Contact FIG Hotline 415/962-8653

**Orange County Chapter**
Monthly, 3rd Sat., 12 noon, Fullerton Savings, 18020 Brockhorst, Fountain Valley. 714/896-2016

**San Diego Chapter**
Weekly, Thurs., 12 noon. Call Guy Kelly, 714/268-3100 x4784

### Massachusetts

**Boston Chapter**
Monthly, 1st Wed., 7 p.m. Mitre Corp. Cafeteria, Bedford, MA. Bob Demrow, 617/688-5661 x198

### Michigan

**Detroit Chapter**
Call Dean Vieau, 313/493-5105

### Minnesota

**MNFIG Chapter**
Monthly, 1st Mon. Call Mark Abbot (days) 612/854-8776 or Fred Olson, 612/588-9532, or write to: MNFIG, 1156 Lincoln Ave., St. Paul, MN 55105

### New Jersey

**New Jersey Chapter**
Call George Lyons, 201/451-2905 eves.

### New York

**New York Chapter**
Call Tom Jung, 212/746-4062

### Oklahoma

**Tulsa Chapter**
Monthly, 3rd Tues., 7:30 p.m., The Computer Store, 4343 So. Peoria, Tulsa, OK. Call Bob Giles, 918/599-9304 or Art Gorski, 918/743-0113

### Oregon

**Portland Chapter**
New Chapter! Call Timothy Huang, 9529 Northeast Gertz Circle, Portland, OR 97211, 503/289-9135

### Pennsylvania

**Philadelphia Chapter**
New Chapter! Call Barry Greebel, Continental Data Systems, 1 Bala Plaza, Suite 212, Bala Cynwid, PA 19004

### Texas

**Austin Chapter**
Call John Hastings, 512/327-5864

**Dallas/Ft. Worth Chapter**
Monthly, 4th Thurs. 7 p.m., Software Automation, 1005 Business Parkway, Richardson, TX. Call Marvin Elder, 214/231-9142 or Bill Drissel, 214/264-9680

### Utah

**Salt Lake City Chapter**
Call Bill Haygood, 801/942-8000

### Vermont

**ACE Fig Chapter**
New Chapter! Monthly, 4th Thur., 7:30 p.m., The Isley Library, 3rd Floor Meeting Rm., Main St., Middlebury, VT 05753. Contact Hal Clark, RD #1 Box 810, Starksboro, VT 05487, 802/877-2911 days; 802/453-4442 eves.

### Virginia

**Potomac Chapter**
Monthly, 1st Tues. 7p.m., Lee Center, Lee Highway at Lexington St., Arlington, Virginia. Call Joel Shprentz, 703/437-9218 eves.

### Washington

**Seattle Chapter**
Call Chuck Pliske or Dwight Vandenburg, 206/542-7611

## FOREIGN

### Australia

**Australia Chapter**
Contact Lance Collins, 65 Martin Rd., Glen Iris, Victoria 3146, or phone (03) 292600

### Canada

**Southern Ontario Chapter**
Contact Dr. N. Solnseff, Unit for Computer Science, McMaster University, Hamilton, Ontario L8S 4K1, 416/525-9140 x2065

**Quebec Chapter**
Call Gilles Paillard, 418/871-1960 or 643-2561

### England

**English Chapter**
Write to FORTH Interest Group, 38 Worsley Rd., Frimley, Camberley, Surrey, GU16 5AU, England

### Japan

**Japanese Chapter**
Contact Masa Tasaki, Baba-Bldg. 8F, 3-23-8 Nishi-Shimbashi, Minato-ku, Tokyo, 105 Japan

### Netherlands

**HCC-FORTH Interest Group Chapter**
Contact F.J. Meijer, Digicos, Aart V.D. Neerweg 31, Ouderkerk A.D. Amstel, The Netherlands

### West Germany

**West German Chapter**
Contact Wolf Gervert, Roter Hahn 29, D-2 Hamburg 72, West Germany, (040) 644-3985

## SPECIAL GROUPS

**Apple Corps FORTH Users Chapter**
Twice monthly, 1st & 3rd Tues., 7:30 p.m., 1515 Sloat Blvd., #2, San Francisco, CA. Call Robert Dudley Ackerman, 415/626-6295

**Nova Group Chapter**
Contact Mr. Francis Saint, 2218 Lulu, Witchita, KS 67211, 316/261-6280 (days)

**MMSFORTH Users Chapter**
Monthly, 3rd Wed., 7 p.m., Cochituate, MA. Dick Miller, 617/653-6136

The following vendors have versions of FORTH available or are consultants. (FIG makes no judgment on any products.)

**ALPHA MICRO**

Professional Management Services
724 Arastradero Rd. #109
Palo Alto, CA 94306
(408) 252-2218

Sierra Computer Co.
617 Mark NE
Albuquerque, NM 87123

**APPLE**

IDPC Company
P. O. Box 11594
Philadelphia, PA 19116
(215) 676-3235

IUS (Cap'n Software)
281 Arlington Avenue
Berkeley, CA 94704
(415) 525-9452

George Lyons
280 Henderson St.
Jersey City, NJ 07302
(201) 451-2905

MicroMotion
12077 Wilshire Blvd. #506
Los Angeles, CA 90025
(213) 821-4340

**CROSS COMPILERS**

Nautilus Systems
P.O. Box 1098
Santa Cruz, CA 95061
(408) 475-7461

**polyFORTH**

FORTH, Inc.
2309 Pacific Coast Hwy.
Hermosa Beach, CA 90254
(213) 372-8493

LYNX
3301 Ocean Park #301
Santa Monica, CA 90405
(213) 450-2466

M & B Design
820 Sweetbay Drive
Sunnyvale, CA 94086

**Micropolis**

Shaw Labs, Ltd.
P. O. Box 3471
Hayward, CA 94540
(415) 276-6050

**North Star**

The Software Works, Inc.
P. O. Box 4386
Mountain View, CA 94040
(408) 736-4938

**PDP-11**

Laboratory Software Systems, Inc.
3634 Mandeville Canyon Rd.
Los Angeles, CA 90049
(213) 472-6995

**OSI**

Consumer Computers
8907 LaMesa Blvd.
LaMesa, CA 92041
(714) 698-8088

Software Federation
44 University Dr.
Arlington Heights, IL 60004
(312) 259-1355

Technical Products Co.
P. O. Box 12983
Gainsville, FL 32604
(904) 372-8439

Tom Zimmer
292 Falcato Dr.
Milpitas, CA 95035

**1802**

FSS
P. O. Box 8403
Austin, TX 78712
(512) 477-2207

**6800 & 6809**

Talbot Microsystems
1927 Curtis Avenue
Redondo Beach, CA 90278
(213) 376-9941

**TRS-80**

The Micro Works (Color Computer)
P. O. Box 1110
Del Mar, CA 92014
(714) 942-2400

Miller Microcomputer Services
61 Lake Shore Rd.
Natick, MA 01760
(617) 653-6136

The Software Farm
P. O. Box 2304
Reston, VA 22090

Sirius Systems
7528 Oak Ridge Hwy.
Knoxville, TN 37921
(615) 693-6583

**6502**

Eric C. Rehnke
1067 Jadestone Lane
Corona, CA 91720
(714) 371-4548

Saturn Software, Ltd.
P. O. Box 397
New Westminister, BC
V3L 4Y7 CANADA

**8080/Z80/CP/M**

Laboratory Microsystems
4147 Beethoven St.
Los Angeles, CA 90066
(213) 390-9292

Timin Engineering Co.
9575 Genesse Ave. #E-2
San Diego, CA 92121
(714) 455-9008

**PolyMorphic Systems**

Abstract Systems
Lower Prospect Hill
Chester, MA 01011
(413) 354-7750

**Application Packages**

InnoSys
2150 Shattuck Avenue
Berkeley, CA 94704
(415) 843-8114

Decision Resources Corp.
28203 Ridgefern Ct.
Rancho Palo Verde, CA 90274
(213) 377-3533

**68000**

Emperical Res. Grp.
P. O. Box 1176
Milton, WA 98354
(206) 631-4855

**Firmware, Boards and Machines**

Datricon
7911 NE 33rd Dr.
Portland, OR 97211
(503) 284-8277

Forward Technology
2595 Martin Avenue
Santa Clara, CA 95050
(408) 293-8993

Rockwell International
Microelectronics Devices
P.O. Box 3669
Anaheim, CA 92803
(714) 632-2862

Zendex Corp.
6398 Dougherty Rd.
Dublin, CA 94566

**Variety of FORTH Products**

Curry Assoc.
P.O. Box 11324
Palo Alto, CA 94306
(415) 322-1463

Global Data Aregon
1911 Wright Circle
Anaheim, CA 92806

Interactive Computer Systems, Inc.
6403 Di Marco Rd.
Tampa, FL 33614

Mountain View Press, Inc.
P. O. Box 4656
Mountain View, CA 94040
(415) 961-4103

Supersoft Associates
P.O. Box 1628
Champaign, IL 61820
(217) 359-2112

**Consultants**

Creative Solutions, Inc.
4801 Randolph Rd.
Rockville, MD 20852

Dave Boulton
P.O. Box 1385
Redwood City, CA 94064
(415) 363-8208

Leo Brodie
9720 Baden Avenue
Chatsworth, CA 91311
(213) 341-4313

Go FORTH
504 Lakemead Way
Redwood City, CA 94062
(415) 366-6124

Inner Access
517K Marine View
Belmont, CA 94002
(415) 591-8295

Laxen & Harris, Inc.
24301 Southland Drive, #303
Hayward, CA 94545
(415) 887-2894

Microsystems, Inc.
2500 E. Foothill Blvd., #102
Pasadena, CA 91107
(213) 577-1471

**VENDORS: FORTH DIMENSIONS will go to a product matrix in Volume IV. Send in a list of your products and services as soon as possible.**

IMPORTANT ANNOUNCEMENT...1982 FORML CONFERENCE...Asilomar, California...October 6-8, 1982...IMPORTANT

The 1982 FORML (FORTH Modification Laboratory) is an advanced seminar for the presentation and discussion of FORTH language topics. It is not intended for beginning or casual FORTH programmers. Topics for presentation include: Programming Methodology, Virtual Machine Implementation, Concurrency, Language and Compiler, and Applications.

Attendance to the FORML Conference is limited. The priority for selection is: 1st, Paper presentors who send in their 100 word abstract by the deadline of August 2, 1982. 2nd, Poster presentors who send in their 100 word abstract by the deadline of August 2, 1982. 3rd, FORTH programmers who wish to attend only. Depending upon the response for paper and poster sessions, there may or may not be room for non-presentors.

---

REGISTRATION FORM

Complete and return with a check to FORML to: (Check must be in US funds on a US bank.) FORML, PO Box 51351, Palo Alto, CA 94303. DEADLINE FOR REGISTRATION: July 15, 1982.

NAME_____

COMPANY_____

ADDRESS_____

CITY_____ STATE_____ ZIP_____ COUNTRY_____

PHONE (Day)_____ (Evening)_____

Accommodations: Prices include room, meals, workbook, facilities use and FORML functions. I want:

___Myself in a ___Double (2 Persons) room, $175.00; ___Single, $225.00. My Wife/Husband/Friend will be with me ___, $150.00 room and meals only. I will arrive the day before, Tuesday, Oct 5, please reserve a room for: #___ at $50.00 each double or $60.00 single for a total of $_____. (Dinner on Tuesday and breakfast and lunch on Wednesday included.)

I have been programming in FORTH for: (Years)_____ (Months)_____. I expect to: ___Present a paper, ___Present a poster session, ___Chair a section, ___Be a non-presentor. My topic will be_____

_____

FOR MORE INFORMATION CALL: ROY MARTENS (415) 962-8653

---

## FORTH INTEREST GROUP MAIL ORDER

| | USA | FOREIGN AIR |
|---|---|---|
| ☐ Membership in FORTH INTEREST GROUP and Volume IV of FORTH DIMENSIONS (6 issues) | $15 | $27 |
| ☐ Volume III of FORTH DIMENSIONS (6 issues) | 15 | 18 |
| ☐ Volume II of FORTH DIMENSIONS (6 issues) | 15 | 18 |
| ☐ Volume I of FORTH DIMENSIONS (6 issues) | 15 | 18 |
| ☐ fig-FORTH Installation Manual, containing the language model of fig-FORTH, a complete glossary, memory map and installation instructions | 15 | 18 |
| ☐ Assembly Language Source Listing of fig-FORTH for specific CPU's and machines. The above manual is required for installation. Check appropriate boxes. **Price per each.** ☐ 1802 ☐ 6502 ☐ 6800 ☐ 6809 ☐ 8080 ☐ 8086/8088 ☐ 9900 ☐ APPLE II ☐ PACE ☐ NOVA ☐ PDP-11 ☐ ALPHA MICRO | 15 | 18 |
| ☐ "Starting FORTH" by Brodie. BEST book on FORTH. (Paperback) | 16 | 20 |
| ☐ "Starting FORTH" by Brodie. (Hard Cover) | 20 | 25 |
| ☐ PROCEEDINGS 1980 FORML (FORTH Modification Lab) Conference | 25 | 35 |
| ☐ PROCEEDINGS 1981 FORTH University of Rochester Conference | 25 | 35 |
| ☐ PROCEEDINGS 1981 FORML Conference, Both Volumes | 40 | 55 |
| ☐ Volume I, Language Structure | 25 | 35 |
| ☐ Volume II, Systems and Applications | 25 | 35 |
| ☐ FORTH-79 Standard, a publication of the FORTH Standards Team | 15 | 18 |
| ☐ Kitt Peak Primer, by Stevens. An indepth self-study primer | 25 | 35 |
| ☐ BYTE Magazine Reprints of FORTH articles, 8/80 to 4/81 | 5 | 10 |
| ☐ FIG T-shirts: ☐ Small ☐ Medium ☐ Large ☐ X-Large | 10 | 12 |
| ☐ Poster, Aug. 1980 BYTE cover, 16 x 22" | 3 | 5 |
| ☐ FORTH Programmer Reference Card. If ordered separately, send a stamped, addressed envelope. | | FREE |
| TOTAL | $_____ | |

NAME_____ MAIL STOP/APT_____

ORGANIZATION_____ (if company address)

ADDRESS_____

CITY_____ STATE_____ ZIP_____ COUNTRY_____

VISA #_____ MASTERCARD #_____

EXPIRATION DATE _____ (Minimum of $10.00 on charge cards)

Make check or money order in US Funds on US bank, payable to: **FIG.** All prices include postage. **No purchase orders without check.** California residents add sales tax.

**ORDER PHONE NUMBER: (415) 962-8653**

**FORTH INTEREST GROUP** **PO BOX 1105** **SAN CARLOS, CA 94070**


# FORTH INTEREST GROUP

P.O. Box 1105

San Carlos, CA 94070


BULK RATE
U.S. POSTAGE
PAID
Permit No. 261
Mt. View, CA

+77 94086SITH@R@@@@P
R L SMITH
ESL SUBSIDIAR OF TRW
PO BOX 510
SUNNYVALE, CA 94086

**Address Correction Requested**