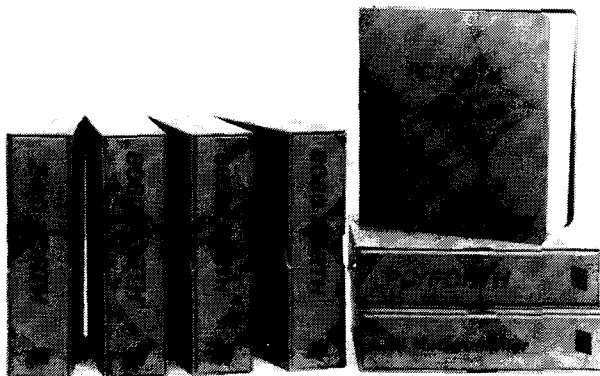


TOTAL CONTROL with LMI FORTH™



For Programming Professionals: an expanding family of compatible, high-performance, Forth-83 Standard compilers for microcomputers

For Development:

Interactive Forth-83 Interpreter/Compilers

- 16-bit and 32-bit implementations
- Full screen editor and assembler
- Uses standard operating system files
- 400 page manual written in plain English
- Options include software floating point, arithmetic coprocessor support, symbolic debugger, native code compilers, and graphics support

For Applications: Forth-83 Metacompiler

- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 8086, 68000, 6502, 8051, 8096, 1802, and 6303
- No license fee or royalty for compiled applications

For Speed: CForth Application Compiler

- Translates "high-level" Forth into in-line, optimized machine code
- Can generate ROMable code

Support Services for registered users:

- Technical Assistance Hotline
- Periodic newsletters and low-cost updates
- Bulletin Board System

Call or write for detailed product information and prices. Consulting and Educational Services available by special arrangement.

LMI Laboratory Microsystems Incorporated
Post Office Box 10430, Marina del Rey, CA 90295
Phone credit card orders to: (213) 306-7412

Overseas Distributors.

Germany: Forth-Systeme Angelika Flesch, Titisee-Neustadt. 7651-1665
UK: System Science Ltd., London, 01-248 0962
France: Micro-Sigma S.A.R.L., Paris, (1) 42.65.95.16
Japan: Southern Pacific Ltd., Yokohama, 045-314-9514
Australia: Wave-onic Associates, Wilson, W.A., (09) 451-2946

Conveniently, both the base of the mask and its binary representation are displayed. (Remember, the sixteen bits are numbered zero through fifteen.)

The word **SLA** in **MASK** is my system's **ML** shift-left arithmetic word (n1 cnt -- n2). Replace it with your appropriate instruction. The **1 OR** in **MASK** takes care of the zero bit position, as in **0 BIT-MASK**.

Forth Dimensions and its contributors often supply me with either some finishing touches or an idea to expand on. Thanks!

Sincerely,

Gene Thomas
Little Rock, Arkansas

Student Roots

Dear Editor,

During this Summer Quarter of 1986, I have been providing the coursework for a student taking "Forth Programming" at Auburn University at Montgomery. As one of his assignments, this student (Hunter Moseley) was required to write a square root in Forth (F83) based upon a Newton's method-type algorithm. However, Hunter went beyond my thought and wrote code that put mine to shame. My code is shown in Figure One.

The **D*/** used does the same thing as ***/** but with double-precision numbers. In other words, (d1 d2 d3 -- d4). Also, the **2NIP** is a double-precision **NIP**. I hated to use the double-precision words, but for the accuracy needed, they were necessary.

Hunter's code was simply that shown in Figure Two.

In a time test on a Zenith-151 with 10,000 iterations, dropping the result each time, Hunter's code guaranteed 119 seconds with any input from zero to 32,766. Mine, however, with an equivalent range of inputs, does the square root of one in seventy-five seconds, the square root of two in 280 seconds, and gets even worse after that.

As can be seen, the two approaches are based on the same idea, but Hunter's does no bound checking. His

Davies' Square Roots

```
: SQRT ( d1 d2 -- d3 )
  RECURSIVE
    2OVER 2OVER 2OVER 10000 0 2ROT D*/ 2SWAP
    2OVER 10000 0 2SWAP D*/ 2OVER D- DABS
    5 0 D< IF 2NIP 2NIP EXIT
      ELSE D+ D2/
      THEN
    SQRT ;

: SQR ( n1 -- n2 )
  10000 *D 10000 0 SQRT 10000 UM/MOD NIP ;
```

Figure One

```
: SQR ( n1 -- n2 )
  1 10 0 DO 2DUP / + 2/ LOOP NIP ;
```

Figure Two

```
CODE SQR ( n1 -- n2 )
  DX POP SI PUSH DX SI MOV 1 # BX MOV
  10 DO
    DX DX XOR SI AX MOV BX DIV AX BX ADD BX SAR
  LOOP
  SI POP BX PUSH NEXT END-CODE
```

Figure Three

```
: DSQR ( d1 -- d2 )
  1. 19 0 DO 2OVER 2OVER D/ D+ D2/
    LOOP 2SWAP 2DROP ;
```

Figure Four

simpler application of the algorithm is much slicker — beauty in Forth.

Additionally, as an experiment with F83's assembler, I translated Hunter's algorithm into assembly. The code is listed in Figure Three. A time test on the Zenith-151 with 10,000 iterations, dropping the result each time, guaranteed five seconds! Yes, that's right — 2,000 iterations per second! Perhaps this amazes no one else, but I was somewhat shocked.

For those interested, Hunter also has the signed, double-precision version of the square root. The code is in Figure

Four. The **D/** is a double-precision divide. If anyone is interested in the code for these operators and their double-precision primitives, I will gladly share them.

In any case, I present these attempts as examples of how traditional mathematical thought sometimes must give way to the more efficient patterns used by our friends — the computers — and Forth.

Sincerely yours,

R.L. Davies
Montgomery, Alabama

Second Take:

Multiple LEAVES by Relay

Dear Mr. Ouverson:

Please discard my previous letter to you (*Forth Dimensions VIII/3*), as it was completely erroneous. My intended verification test wound up with confusion between the fig-FORTH words in my system and the new words, due to my carelessness! Here is the new manuscript:

John Hayes' "Another Forth-83 LEAVE" (VII/1) stimulated me to try to find an even simpler way to handle multiple Forth-83 LEAVES. I decided that a straight-forward approach involved having each LEAVE simply branch to the next LEAVE, with the last one removing the index values from the return stack and branching to the word following LOOP.

I "grafted" such a construction onto fig-FORTH with the definitions below; words with a * prefix are used to identify changes from fig-FORTH. Unstarred words such as (DO) and (LOOP) are unchanged. Whenever a *LEAVE is compiled, the variable PLACE is used to hold the location of its branch value for later adjustment. This variable also serves as a flag to show that there is a leave branch to be resolved. *LOOP calls a >RESOLVE to install the jump value of the preceding (if any) *LEAVE; also, if there is a *LEAVE in the word, a special OUTLEAVE is compiled immediately following the (LOOP) closure. OUTLEAVE removes the (two) loop parameters from the return stack and proceeds to the next word, i.e., the word that was entered after *LOOP. If the *LEAVE command is not invoked at run time, the normal loop operation removes these parameters from the return stack, so OUTLEAVE must be skipped over. *LOOP compiles this bypass with a BRANCH 4 which is encountered in normal loop completion. Alternatively, (LOOP) could be modified to use OUTLEAVE in normal loop completion.

Note that OUTLEAVE can be a primitive which removes two words from the return stack by using PLA four times. If OUTLEAVE is defined as a