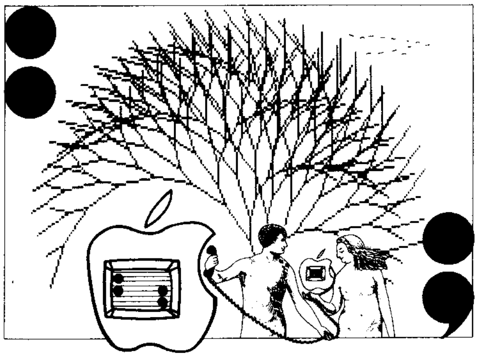
FORTH Dimensions

Volume 5, Number 4
November/December 1983
$2.50



Multi-Tasking

FEATURES

DEPARTMENTS

FORTH Dimensions

Published by FORTH Interest Group
Volume V, No. 4
November/December 1983
Editor
Marlin Ouverson
Publisher
Roy C. Martens
Typesetting/Production
LARC Computing, Inc.
Cover Art
Al McCahon

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.

Letters to the Editor

A Friend in Need

Sirs:

I need help. I have used my computer (Kaypro 10) for word processing only. I was not interested in learning a language till I began reading about Forth in the journals. I picked up *Starting Forth* and leafed through it. I said to myself, "I can learn this!" I bought the book and ordered the language disk. The computer store ordered SL5 from SuperSoft. I started to learn the language using *Starting Forth* as my documentation. I soon discovered that they were not mated to one another. Valiantly, I struggled with the six pages of SuperSoft tutorial, then back to *Starting Forth*, trying to change the words that wouldn't work. After many hours, I gave up. After all that, here is my question:

What should I buy, as a beginner, that would let me use *Starting Forth* as my documentation?

Awaiting your information, I am
Yours sincerely,

Duane Windemiller
367 Ocean Boulevard
Hampton Beach, NH 03842

Breakpoint Revisited

Dear Editor,

The breakpoint tool described in *Forth Dimensions* (Vol. V, No. 1)

(Continued)

Editorial

Fifth Forth Fest

Another October has come and gone, and with it the Fifth Annual Forth Convention. Hats off to the FIG board of directors, especially to Robert Reiling and Gary Feierbach, who created an informative, streamlined event for the 1200 attendees and the thirty exhibitors. It is a sign of the maturation of the Forth community and of the diligence of the organizers that the convention earned coverage in the public media as well as in the trade press.

At each of the annual conventions, one FIG member is named "Figgie of the Year." The recipients are those who have made exceptional contributions to Forth and its growth in the industry. This year, the whimsy of the title was surpassed only by the surprise of John D. Hall when his name was announced. John's work as coordinator of local chapters of the Forth Interest Group has been diligent, thorough and unselfish. Through his efforts, many chapters have been guided into formation and now serve the world-wide Forth programming community. This has greatly enhanced Forth's growth and status. Thanks, John, and congratulations!

Mountain View Press, a major vendor of Forth products, presented a prize at the convention this year. The company had held a contest for those who receive its newsletter. The challenge was to describe Forth in twenty-five words or less for non-Forth people. Charles Moore judged the entries and had this to say before announcing the winner: "Forth resists analysis. It's a right-side of brain function .... Language is practical. Forth is a language, not an operating system." The winner of the contest was Michael Ham of Iowa City, Iowa. His entry reads, "Forth is like the Tao: it is a Way, and is realized when followed. Its fragility is its strength; its simplicity is its direction."

Your editor spent the two-day meeting with authors, authors-to-be and readers. I hope we learned a lot about each other. From what I saw and heard, all of us can look forward to a great deal of exciting material in upcoming issues of *Forth Dimensions*. As the magazine grows, so does the number of ways in which we can serve you. Especially vital is that this publication always remain an open forum. There must always be room for you to voice your support and concerns, not just of the magazine, but about the language and community in which we are immersed.

A small commercial: the ways we can serve you are limited only by our resources. Help *Forth Dimensions* and the Forth programming community to grow by joining as an individual member of the Forth Interest Group. Sure, it's nice to share reading materials. But why not get your own membership and begin sharing this magazine with new people who haven't yet been exposed to Forth. Besides, aren't you tired of getting that well-thumbed issue after everyone else has read it?

In the next issue we will bring tidings from the November FORML conference. Until then, accept our heartfelt wishes for a pleasant holiday season and a new year of peace.

—Marlin Ouverson
Editor

needs to be slightly modified for versions of Forth, such as MVP FORTH, that use a vectored **INTERPRET**. This results in three levels of return instead of two in going into and leaving the new interpreter. In the enclosed code, I used **RESUME** because **GO** has a different function in MVP FORTH (it is a code word that permits one to directly load the program counter). The changes are to replace the four in line seven with six, and to add an additional **R> DROP** in line eleven (line numbers refer to the original listing on page nineteen of *Forth Dimensions*). The version for my Forth is shown in figure one.

File Fan

Dear Sir:

After laying off for a couple of years, I have rejoined FIG; and I just treated myself to an (extended) evening of catching up on Volumes III and IV of *Forth Dimensions*. So many (mostly pleasurable) reactions are reverberating in my mind that it has taken an act of utmost discipline to restrict my comments to the few below.

First, let me say that I am *not* a Forth fanatic. I have been driven somewhat reluctantly to Forth after careful study of a number of languages for personal computing, including, most recently, C.

The most important issue on my mind is that of operating systems. I see, over a two-year period, some significant evolution away from the Forth screen system and towards the file handling systems of the host environment, *e.g.* CP/M. This evolution is being driven by the vendors, and my impression is that it is being quietly resisted by FIG.

As a personal computerist, I do not see this as primarily an issue of being able to share disk space with the resident system. Rather, for me, it is simply an issue of being able to use with Forth the power of whatever operating system is available.

Editors and word processors are among the programmer's most personal tools. On the face of it, there just can be no comparison between a Forth screen-based editing system and a good file-based system. On top of that, there is a whole industry out there devoted to developing editors and word processors; and many of us own several examples of each, some of which we find extremely useful.

The design of my personal Z-80, CP/M Forth system includes file variables for sequential and random CP/M files, and allows an arbitrary number of open channels. It also includes a file control disk stack which makes it possible for any Forth source file to load another, to any depth, and pick up where it left off. Some of the possibilities this opens up for the organization of libraries should be clear.

Let me make a radical proposal. Abandon the screen system and, in

(Continued on page 28)

```
 OK
162 LIST
SCR #162
  0 ( BREAK & GO (RESUME) FORTH DIMENSIONS VOL 5 # 1 )
  1  VARIABLE CHECK   ( COMPILE BREAK INTO : DEF )
  2 : BREAK CR ." BREAK S= " .S CR ." R= " ( R.N )
  3   RP@ 6 - CHECK ! 0 BLK ! BEGIN QUERY INTERPRET ." aok " CR
  4   AGAIN ;
  5 : RESUME ( GO IN FD ) RP@ CHECK @ = IF R> DROP R> DROP R> DROP
  6   ELSE ." can't resume " QUIT THEN ;
  7
  8
  9
 10
 11
 12
 13
 14
 15
OK
```

Figure One

Why Novices Use So Many Variables

Michael Ham
Iowa City, Iowa

Forth programmers often find themselves tutoring Forth novices. The more experienced the tutors, the harder it is for them to recall their own early difficulties and to understand the sources of the beginner's problems. Without knowing the source of the problem, one cannot attack it at its root, and instead must correct the errors one by one, as they occur. This paper discusses a possible source of a common novice error: using unnecessary variables.

I recently found a conceptual block when I examined the reasons I had coded a program awkwardly (see listing). The listed definition works — it does, in fact, display the primes less than 1000 — but an experienced Forth programmer will instantly see that some changes are in order:

1. Eliminate the variable **PRIME**.
2. Replace **2 PRIME !** in line five with **1**.
3. Replace **0 PRIME !** in line eight with **DROP 0**.
4. Eliminate **1 PRIME !** in line nine.
5. Eliminate line eleven altogether.

```
 0 ( EXAMPLE OF BEGINNER CAUTION                 M Ham          8/23/83 )
 1
 2   VARIABLE PRIME ( START zeroes system clock; TIME prints time)
 3
 4 : PRIMES   ( -- )  CR START
 5           1001 1 DO   2 PRIME !
 6                       501 2 DO   J I >
 7                                  J I MOD O=
 8                                  AND     IF 0 PRIME ! LEAVE THEN
 9                                  J I =   IF 1 PRIME ! LEAVE THEN
10                             LOOP
11                       PRIME @ DUP 2 = IF DROP 1 THEN
12                       IF I 5 .R THEN
13                  LOOP
14           TIME ;
15
```

Listing

The word still works with these changes, but more efficiently. (The actual difference in execution time is one-tenth of a second on my Forth: 29.0 vs. 28.9 seconds.) What caused the superfluous code?

I discovered the redundancies by chance: I picked up the listing two or three days after writing it, and on glancing at it, suddenly saw that the careful replacement of the two by a one (line eleven) was unnecessary, since two would serve as a true flag as well as one.

But why had I even put a two into **PRIME** in the first place? Its origin seemed to lie in my unexamined feeling that, on coming out of the loop, it would be good to know how I discovered the nature of the number: from within the loop (**PRIME** contains zero or one), or by exhausting all possible divisors (**PRIME** contains two). Perhaps I also had wanted to avoid declaring, before even starting the loop, whether the number was prime or not: two was a way of not taking a position.

I decided that I might as well leave the two on the stack as a true flag — and, given that, no reason it shouldn't be a one from the start. So I dropped

DUP 2 = IF DROP 1 THEN

from line eleven, and in line five initialized **PRIME** with **1 PRIME !**.

Then I realized that when exiting the loop early (via one of the **LEAVE**s), I was fetching from **PRIME** the value just put into it. What was going on? Looking at the code again, I saw that **PRIME** was unnecessary, and that all the changes listed above should be made.

Why had I created **PRIME** in the first place? I thought about it and concluded that I had not fully understood Brodie's cautionary remark in *Starting Forth* (page ninety-three) about making the stack effect of a word be the same, regardless of which part of the word was executed. I had thought that he meant you should be careful about the stack effect only to prevent stack underflow or overflow, which would crash the program.

It is that, of course, but with this example I saw something more — something undoubtedly so familiar to those who are used to Forth and the stack that they accept it unthinkingly: if words keep properly to themselves, using the stack only for their expected input and output, and cleaning up after themselves, then they can be looked on as sealed systems, having no effect on anything that might already be on the stack but outside their sphere of influence.

In my prime number routine I had pulled the flag off the stack and tucked it into a variable to keep it safe. No telling *what* might happen with all that thrashing about on the stack. The flag might get hit by a wild shot or ricochet. So off it went for safekeeping, to be fetched from its cubbyhole when needed.

I immediately recalled another example of the same behavior in a program to compute quartiles. The program collects the scores for each group, sorts them, computes the quartiles and prints the results, cycling in an endless loop to get the figures for the next group. After every eight groups I wanted to do a form feed to avoid printing on the perforation. At first I had put the count of the groups in a variable, once again to keep it safe from all the activity taking place on the stack: gathering data, sorting, computing, printing.

But then I had seen that the sequence of words constituted a closed system, and the count could rest safe and sound on the stack the whole time, totally unaware of the storm of activity raging right over its head. The count could go on the stack at the beginning of the loop, and remain through the complete routine for each group. At the end of each, the count would emerge, back on top of the stack, not a hair out of place.

It still somewhat amazes me that merely by making sure words practice good stack hygiene you can be so confident about what is happening on the stack. If you investigate the unnecessary variables that beginning Forth programmers use, I bet you'll find them unaccustomed to using a stack and nervous about unforseen by-products of all that frantic stack activity. They become overprotective and use a mechanism familiar from their experience with other programming languages to shield their counts and intermediate results from possible harm. They see the whirlwind of activity that will descend on the stack, but they fail to recognize that it is harmless, enclosed within the inviolable sphere defined by the words' net stack effect.

You need only convince novices that, if they write their definitions with a careful eye on the stack effects, they can drop enormous clusters of words on top of a lonely little number on the stack and, after the words have done their work and gone and the dust has settled, that number will stand there, once more at the top of the stack, absolutely unharmed.

Yet Another Number Utility

David McKibbin
Timonium, Maryland

In any program or environment, there is typically one number base that predominates. Just as typically, there are also times when another number base for input or output would be clearer or easier. The following word set allows for these single occurrences of numbers which are not in the current number base and can be used in either compile or execution mode.

There is one defining word (**BASE.**) for the class of words that output in the various number bases, and another (**BASE'**) for the class of words that input. I still use <**BUILDS** in my system as the complement to **DOES**>. For those who don't, replace the <**BUILDS** with **CREATE**. Words defined by **BASE.** and **BASE'** have their new, temporary base stored in their parameter fields. When executed, they simply fetch the current number base and save it on the return stack, set **BASE** to the new value stored in their parameter field, perform the input or output, and finally restore the old base from the return stack. Additionally, words defined by **BASE'** are made immediate so that they can be used inside colon definitions. For me, this has been very useful inside **DO...LOOP** where the count is clearer if expressed in decimal but the body of the loop contains bit masks, *etc.*, that are more clearly expressed in hex.

Inside **BASE'** the code fragment

0 0 BL WORD CONVERT 2DROP

is used to get a sixteen-bit number from the input device to the stack. Then **LITERAL** is used either to do nothing (in execution mode) or compile the number as a literal (in compile mode). This is presuming upon the "state smartness" of **LITERAL**. With the current move away from state smartness, I recommend checking your implementation of **LITERAL**.

In writing programs, I will usually leave the current number base in **DECIMAL** and use these words for any departures from that. This has solved several recurring problems. First, when I see a number, I know that it is in decimal if not preceded by a modifier. I don't have to reverse-scan the code looking for the last base change. Second, I can painlessly insert hex bit masks regardless of the current number base. And third, I can enter numbers in whatever base most clearly conveys my intent or purpose without bulking up the code with explicit **HEX**, **DECIMAL** or **OCTAL** words.

```
SCR # 30
( NUMBERS -  BASE., B., O., X., H.                 DTM  03Aug81 )

: BASE.                                  ( base print defining word *)
    <BUILDS  ,   DOES>  BASE @ >R  @ BASE !  U.  R> BASE !  ;

 2 BASE. B.                                      ( binary  print *)
 8 BASE. O.                                      ( octal   print *)
10 BASE. X.                                      ( decimal print *)
16 BASE. H.                                      ( hex     print *)
```

```
SCR # 31
( NUMBERS - BASE', B', O', X', H'                   DTM  25Feb82 )

: BASE'
    <BUILDS ,   IMMEDIATE
    DOES>  BASE @ >R  @ BASE !
           0 0  BL WORD  CONVERT 2DROP  [COMPILE] LITERAL
           R> BASE !  ;

 2 BASE' B'                                      ( binary  input *)
 8 BASE' O'                                      ( octal   input *)
10 BASE' X'                                      ( decimal input *)
16 BASE' H'                                      ( hex     input *)
```

```
OK
DECIMAL 1234 OK
DUP B. 10011010010 OK
DUP O. 2322 OK
DUP X. 1234 OK
DUP H. 4D2 OK
. 1234 OK
H' 1234  . 4660 OK
X' 1234  . 1234 OK
O' 1234  . 668 OK
B' 1000101  . 69 OK
: strip-parity  H' 7F AND  ; OK
193 strip-parity  . 65 OK

00:00:00  00/00/00            Sygnetron FORTH  -  revision 2.2
```

Manufacturing Cost Program

Marc Perkel
Springfield, Missouri

This program demonstrates how simply cost analysis can be done in Forth. In this example, Forth is used not only as a compiler, but as a job control language. In other words, source screens do not have to contain code to be compiled. They may contain lists of commands to be executed. This execution replaces typing the same commands from the keyboard. In this way, disk files of command strings to be executed can be changed at will.

What the Program Does

This program calculates the cost of manufacturing a fruit basket. The user first compiles screen eighty, which is the program. Then the user types the word **PRICES**, which causes the prices to compile. Then the user types **FIGURE FRUIT-BASKET**, which causes the cost of the fruit basket to be printed on the screen. Any time the user wants to change prices, he brings in the editor and types over the old data. Or, if the user wants to change the materials, he likewise types in the new materials by using the editor.

```
Screen 80 (*) 128
    ( Pricing Program )  : TASK ;  DECIMAL

    2VARIABLE TOTAL

    : .MONEY   <# # #  ASC . HOLD  #S  ASC $ HOLD  #> TYPE SPACE ;
    : +TOTAL   TOTAL 2@ D+ TOTAL 2! ;
    : PRICE    CREATE , , DOES> 2@ >R OVER U* ROT R> * + +TOTAL ;
    : COST     CREATE , , DOES> 2@ +TOTAL ;
    : LOCATES CREATE , DOES> @ LOAD ;
    : DOZEN    12 * ;
    : FIGURE   0. TOTAL 2! [COMPILE] ' EXECUTE  TOTAL 2@ .MONEY ;

    ( Directory )
    129 LOCATES PRICES
    130 LOCATES FRUIT-BASKET


Screen 81 (*) 129
    ( Friut Basket Costs )

    .34  PRICE  APPLES
    .26  PRICE  BANANAS
    .47  PRICE  ORANGES
    .03  PRICE  CHERRIES
    .54  PRICE  GRAPEFRUIT
    .32  PRICE  PEARS
    .29  PRICE  PEACHES
    .02  PRICE  GRAPES

    2.45  COST  BASKET
    1.08  COST  PACKAGING
    3.67  COST  SHIPPING
    1.00  COST  HANDLING


Screen 82 (*) 130
    ( Fruit Basket materials )

    5 PEARS
    8 ORANGES
    3 GRAPEFRUIT
    2 DOZEN CHERRIES
    3 DOZEN GRAPES
    6 PEACHES
    4 APPLES
    7 BANANAS

    BASKET
    SHIPPING
    PACKAGING
    HANDLING
```

How the Program Works

The variable **TOTAL** accumulates the total costs. **.MONEY** is used to print the total in dollars and cents format. **+TOTAL** adds the thirty-two-bit number on top of the stack to **TOTAL**. **PRICE** is a defining word; at compile time, it creates a definition (containing the price) that, when executed, multiplies the price by the number on the stack and adds it to the total. For example, **34 PRICE APPLES** creates the word **APPLES** and assigns the value .34 each. When **4 APPLES** executes, **APPLES** multiplies four times thirty-four cents and adds it to **TOTAL**.

COST works like **PRICE** except that it doesn't multiply. It assumes a quantity of one. **DOZEN** merely multiplies the quantity by twelve. **LOCATES** is a defining word used here as a crude disk directory. **LOCATES** creates a word (*e.g.* **PRICES**) and stores 129 into it. When **PRICES** executes, it loads screen 129. The word **FIGURE** executes the following word first and displays the **TOTAL**.

The interesting thing to note is that the words created by **PRICE, COST** and **LOCATES** become part of the Forth vocabulary. Thus, typing any word created by **LOCATES** will cause a predetermined screen to load. Any word created by **COST** will cause an amount to be added to **TOTAL**, and likewise with **PRICES**. The Forth outer interpreter is used as a big, text-driven case statement and eliminates the need to write one as part of the program.

In conclusion, I challenge anyone to write such an elegant program in any other language. This program is in use by a local manufacturer, and I hope others will expand on this unusual technique in real-world applications.

Menu-Driven Software

John Bowling
Phoenix, Arizona

Menu-driven software has always been easier for all but the most sophisticated of users. The programmer puts a list on the screen with a brief description of the option, along with a key code. The user enters the simple key code, and is off and running in a new section of the program. If the new section has options, up comes another menu. If the purpose of the software requires it, a menu tree can have hundreds of levels, with some menus able to jump tens of levels per key code.

Menus make a user's job very easy, but can be a headache for a programmer. They require that the software write an entire page out to a terminal, pick up a user's response, index into a vector table, and jump to a new section of code. If the software is very large, overlays or some type of virtual memory scheme may be required. Ideally, the language selected should support menus without having to code each one separately. Using a subroutine for the menu code requires passing a pointer to the character data for the terminal, and a pointer to the vector table. Somewhere in memory reside pages of text for the terminal and more pages of vector tables.

There is a simple solution to the problem in Forth. The Forth listing shown here is quite simple, and is designed to work with any standard FIG-Forth system. The two primary words, **M"** and **MENU-UP**, are supported by a mode variable (**MUMODE**), a pointer variable (**MP**), and two arrays (**MNU1** and **MNU2**). **MENU-UP** controls the use of **M"** by setting **MUMODE** true and sets **MP** to zero.

When a screen is loaded and an **M"** is encountered, **M"** checks **MUMODE**. If **MUMODE** is true, two numbers are taken from the input stream (the disk), and saved in the array positions pointed to by **MP**. **MP** is incremented once

```
130
( Sample Directory Screen for MENU      jlb  October 12, 1983 )

DECIMAL                                 ( Header at top of page )
CR  ." Disk Directory" 20 Spaces  ." October 12, 1983"  CR
CR  ." Disk Menu:"     24 SPACES  ." Drive "   DR @ . CR CR

                         ( TYPE out directory according to MUMODE )

M"  12  19 Assembler (6502)"
M" 131 131 Directory Menu"           M"  22  23 Disking"
M"  20  21 Documentor"               M"  25  39 Editor"
M" 180 185 Permanent Extensions"
M"  10  11 Startup Loader"
```

```
131
( Non-Standard words used for MENU      jlb  October 12, 1983 )
FORTH DEFINITIONS  DECIMAL

 0 VARIABLE DR

     ( DR  is a variable that contains the drive number last )
                                   ( specified by DR0 or DR1 )

: DR0  EMPTY-BUFFERS DR0 0 DR ! ;                      ( -- )
: DR1  EMPTY-BUFFERS DR0 1 DR ! ;                      ( -- )

     ( Adjust a number between Low and High values, inclusive )
: LIMITS  ROT MIN MAX ;                       ( n low high --- n )
: PAGE    0C EMIT ;             ( PAGE should clear the screen )

-->
```

```
132
( MENU     variables  ?NUMB             jlb  October 12, 1983 )

DECIMAL                                 ( number storage arrays )

0 VARIABLE MNU1  52 ALLOT    MNU1 52 BLANKS
0 VARIABLE MNU2  52 ALLOT    MNU2 52 BLANKS

( Array Pointer                Mode                             )

0 VARIABLE MP             0 VARIABLE MUMODE

-->
```

```
133
( MENU  M"                                  jlb  October 12, 1983 )

                                         ( Menu display function )
: M"         MUMODE @  IF                             ( Save numbers )
               ( Print out a code letter for the MENU  --   A> )
             MP @ 65 +                            ( Adjust to Alpha )
             . 52 EMIT 2 SPACES          ( Print out Menu number )
             MP @ DUP 2* DUP                ( Adjust array pointer )
     ( Pickup the numbers following M" and place them in array )
             32 WORD HERE NUMBER DROP  SWAP MNU1 + !   ( Pull # )
             32 WORD HERE NUMBER DROP  SWAP MNU2 + !   ( Pull # )
             1+  0 26 LIMITS   MP !  THEN  ( Increment pointer )
             34 WORD HERE COUNT TYPE   CR  ;   ( Type remainder )

-->                      ( M" should be used as you would use ." )


134
( MENU  MENU-UP                             jlb  October 12, 1983 )

             ( Set variables and printout the specified screen )

: MENU-UP    0 MP !  1 MUMODE !   LOAD    0 MUMODE !  CR
                                 ( Request a number from the user )
             ." Your Selection {1 to " MP @ 65 + . ." }? "
                                   ( Adjust number within LIMITS )
             KEY  65 -  0 MP @  1 -  LIMITS   2* DUP
                                 ( Call saved numbers from arrays )
             MNU2 + @   SWAP   MNU1 + @    0 MP ! ;

-->



135
( MENU  Sample uses                         jlb  October 12, 1983 )

              ( Sample uses of MENU              menu -> editor )
: EM         PAGE ." Edit Menu" 130 MENU-UP EDITOR E ;

                                              ( menu -> compiler )
: LM         PAGE ." Load Menu" 130 MENU-UP SWAP DROP LOAD ;

               ( Print out MENU screen without saving numbers )
                                        ( M" works just as ." does )
: DIR        PAGE 0 MUMODE ! 0 MP ! 130 LOAD ;

;S                              ( See text about EDITOR modifications)
    ( Replace all 130's with the screen number you select for )
                                          ( your directory screen )


Starlight-FORTH OSI V s1.10       copyright 1983 by John Bowling
```

for each execution of **M"** when **MUMODE** is true. Following the extraction of the two numbers, a letter corresponding to **MP** value plus sixty-five is printed, followed by > and the remainder of the data in the input stream up to the next occurrence of ".

After the screen load is completed, the user is requested to press a key between A and **MP** value plus sixty-five. This upper letter, and the maximum value in **MP**, is limited by the space available in the arrays, Z or twenty-six. This limit is enforced in **M"** by **MP**, and in **MENU-UP** by the letter keyed in.

There are three sample methods of using the menu function. **DIR** prints the specified screen as if **M"** were equivalent to .". **LM** will load starting at the screen number in **MNU1**, pointed to by the pressed key. **EM** calls the editor after displaying the menu.

I use a full-screen editor which employs control keys and function keys to perform various functions. It uses two variables called **FIRST** and **LAST** that contain the **LIMIT** values for screen numbers that are allowed to be edited. When using **EM**, the values on the directory screen are placed into **FIRST** and **LAST**. Control-Q will move to the previous screen, but not prior to **FIRST**. Control-W will move to the next screen, but not after **LAST**. Function key five (F5) will move to the **FIRST** screen, and F6 moves to the **LAST**. **EDITOR** sets to vocabulary and **E** enters the editor.

(See figures on page 13)

End Listing

```
Disk Directory

Disk Menu:                          Drive 0

A>  Assembler (6502)
B>  Directory Menu
C>  Disking
D>  Documentor
E>  Editor
F>  Permanent Extensions
G>  Startup Loader

Your Selection {A to G}? G


Stack after key press

TOS 10 11  BOS

OK
```

Sample of 130 DIR

```
Disk Directory

Disk Menu:                          Drive 0

 12  19 Assembler (6502)
131 131 Directory Menu
 22  23 Disking
 20  21 Documentor
 25  39 Editor
180 185 Permanent Extensions
 10  11 Startup Loader
OK
```

Sample of 130 MENU-UP

Vocabulary Tutorial, Part II

Evan Rosen
East Setauket, New York

A commonly recognized problem with the **CONTEXT/CURRENT** scheme arises when programming a code definition for some vocabulary (call it **VOCAB**) which is not **FORTH**. Then **ASSEMBLER** is **CONTEXT** and **VOCAB** is **CURRENT**. If some needed data is in a third vocabulary, the programmer has to declare that vocabulary, get the data to the stack, and then declare **ASSEMBLER** again to continue assembling machine code. This problem and its cousins are at the nuisance level.

More serious is the fact that **CONTEXT/CURRENT** makes implementing "front-end" routines like algebraic parsers very difficult. One has only to write out the solution to the quadratic equation in Forth to realize the need for such a utility. Yet, although parsing schemes in Forth have been proposed many times (see the recent one by Stolowitz[1]), few have found their way into widely distributed systems. This is largely attributable to the difficulty of having more than two vocabularies, **CONTEXT** and **CURRENT**, in the search order. For instance, for an algebraic parser's vocabulary, redefined words would include at least the following:
+ - < > / ( )

These words would have to be reached by the search before their counterparts in the **FORTH** vocabulary, since they are redefinitions. In FIG, this means that the parser vocabulary would have to be **CONTEXT**, and the next searched vocabulary would have to be put in **CURRENT**, producing inelegant and misleading code. Accessing any other vocabulary during parsing, for data pickup or another reason, would become awkward to the point of impracticality.

Still other uses for extra, transient vocabularies, such as named, local arguments, which would make life so

```
Action      ~~~   Resulting search list

                  (pos. 0)                                (pos. 4)
                  (first/context)                  (last/protected)

ONLY        ~~~   ONLY        0       0       0       ONLY

      ONLY erases the 10 bytes of search list and then installs
      (pointers to) itself in the first (context) position in the
      list so that it will show as CONTEXT, and at the last
      ("protected") list position so that it won't be pushed off
      the end as more vocabularies are added.  The 0's in the list
      will be skipped during searches.  ONLY will not be searched
      twice since a "twice in a row" detector in -FIND will
      prevent the second search.

FORTH       ~~~   FORTH       0#1     0#2     0#3     ONLY

      We've tagged the zeroes in the search list so you can watch
      which ones disappear.  FORTH places a pointer to itself into
      the first (context) position of the list.  This is the
      normal function of a vocabulary name in virtually all FORTH
      implementations, and is preserved in this one.  That's a
      good sign.

ALSO        ~~~   FORTH       FORTH   0#1     0#2     ONLY

      Here's part of the magic.  ALSO, a new word, moves entries
      0, 1, and 2 of the search list into positions 1, 2, and 3.
      The entry at position 3 is lost.  The ONLY at position 4 is
      left untouched.  FORTH is now safe from overwriting by the
      next vocabulary call.  The effect of ALSO is simply to make
      room for the next vocabulary, in this case ASSEMBLER...

ASSEMBLER   ~~~   ASSEMBLER   FORTH   0#1     0#2     ONLY

      The vocabulary ASSEMBLER is put at the top of the search
      list.  The search order is now ASSEMBLER FORTH (skip) (skip)
      ONLY.

ALSO        ~~~   ASSEMBLER   ASSEMBLER FORTH   0#1   ONLY

      ASSEMBLER is slid down preparatory to adding another
      vocabulary to the list.  Note that 0#2 has been lost.

PARSE       ~~~   PARSE       ASSEMBLER FORTH   0#1   ONLY

      PARSE is added.  We are now prepared to do assembly code,
      and to parse algebraic expressions while doing do.  And
      there's room for another vocabulary.  Note that if two more
      vocabularies were to be added, e.g.  ALSO VOC1 ALSO VOC2
      then FORTH would be pushed off the end.
```

1. Michael Stolowitz, "Algebraic Expression Evaluation in Forth," *Forth Dimensions*, Vol. IV No. 6.

much easier, also have been impeded by the vocabulary search problem.

The ONLY Solution?

Probably the best and most obvious approach to the outlined situation is simply to allow the programmer to specify a search order for as many vocabularies as are required, and be done with it. Indeed, over the last few years many authors have proposed and/or implemented systems which do just this. And the STOIC language, a variant of Forth, has long had a vocabulary stack for the same reason. But no general agreement on specifics has arisen for Forth.

Recently, however, due to some campaigning by Bill Ragsdale on behalf of his own rather clever scheme[2], a welcome concensus on the next milestone in Forth vocabulary structures may be in sight.

(An important thing to remember when promoting ideas to the rag-tag Forth community is to give your code a short, catchy name like **ONLY** and then emblazon the name on your t-shirt at Forth conferences. It also doesn't hurt if the solution is a good one.)

Following is a description and illustration of the form of Ragsdale's vocabulary structure.

At the heart of the **ONLY** scheme are three words: **ONLY**, **ALSO**, and a new version of **-FIND**. The Forth-79 version of the word **VOCABULARY** is used instead of the FIG version. (The difference makes each one stand-alone or "sealed," *e.g.* with its last link field containing zero, similar to the way the vocabulary **FORTH** is in FIG-Forth.) These words and a few bytes of memory appended to the data area of the variable **CONTEXT** are about all the implementation requires to function, though several other programmer-friendly words are included for convenience.

When the variable **CONTEXT** is defined, it is followed by, say, **8 ALLOT** which reserves some additional memory where the list of (pointers to) vocabularies in the search order will reside. With eight more bytes allotted after **CONTEXT**, there is a total of ten, or room enough for five vocabularies in the list at once. Five entries is an arbitrary but practical list size.

At compile time, the new **-FIND** searches through each of the vocabularies in turn, using the usual **(-FIND)** primitive. But the real innovation is in how the search list is set up and manipulated. We examine this process by starting with the word **ONLY**.

ONLY is a very small vocabulary containing only a few words and having an additional feature not shared with other vocabularies: when executed, **ONLY** zeroes the search list and then installs pointers to itself in the first and last positions in the list. (During search, zeroes in the list are skipped over, and duplicate successive entries do not result in duplicate searches.) The first position is the normal data area of the variable **CONTEXT**, and the last position in the search list is a protected one which is not normally altered by anything except **ONLY**. Saying **ONLY** is how one starts a new search list.

The few words in the **ONLY** vocabulary allow a larger list to be built. The main words are: **ONLY** itself, which always allows you to start over; **ALSO**, which makes room for more vocabularies on the list (see figure one); **FORTH**, which adds the Forth vocabulary to the search list; and the null word, a system detail necessary for interpretation of source code.

Obscure? Probably only at first. Figure one shows a walkthrough of a typical sequence.

Now, because appropriate search orders can usually be set up outside a word to be defined, there is no particular need for vocabularies to be immediate any more, and so in a particular implementation you may find that they are not. Beyond this, if you are only programming in the Forth vocabulary, you'll see very little difference in code. However, all sorts of additional programming utilities are made possible with the new structure. Expect to see them coming up soon.

The next and last installment in this vocabularies series will discuss the marked similarity between the vocabulary structure in Forth (especially FIG-Forth) and the "object" structure in the Xerox-developed language, Smalltalk.

2. William F. Ragsdale, "The 'ONLY' Concept for Forth Vocabularies," *Forth Modification Laboratory Conference Papers*, 1982.

Forth Froth

Wil Baden
Costa Mesa, California

Forth is unusual among programming languages in that it uses) and (for the same purpose as natural language. Indeed, "parenthesis" means "remark" in Greek, and) and (are properly called "parenthesis marks," just as we say "quotation mark" and "exclamation mark."

Other analogies with natural language can be made.

In natural language, a noun is the name of a person, place or thing. In Forth, this corresponds to something whose stack diagram is (-- n1,...n*i*). Nouns can be singular or plural; this corresponds to how many values are put on the stack. A proper noun, in Forth, is a constant.

An adjective modifies a noun. In Forth, the stack diagram is (n1,...n*i* -- n1′,...n*i*′). **1+** is an adjective; so are **+** and **COUNT**.

A verb is the name of an action. In Forth the stack diagram is (n1,...n*i* --).

Intransitive verbs use up one value from the stack (*e.g.* **LIST**) and transitive verbs consume more than one value from the stack. The stack element corresponding to n1 is the subject of the verb, the other elements are objects. The value corresponding to n2 is the direct object; any others are indirect objects. A verb without a subject, *e.g.* the stack diagram would be (--), is an impersonal verb; others are personal verbs.

Prepositions must be followed by another word. Some examples in Forth are :, **VARIABLE**, **FORGET** and **'**.

A pronoun takes the place of a noun. In Forth, these are the stack operators, **DUP** and **OVER**. It is convenient to consider the other stack operators as pronouns also, *e.g.* **ROT**, **SWAP**, and **DROP**. (Note that by our previous definitions, **DROP** could also be a verb and the rest could all be adjectives.)

Interjections are the words in a parenthesis.

Conjunctions are used to join words and phrases. The Forth conjunctions are **BEGIN**, **WHILE**, **REPEAT**, **UNTIL**, **IF**, **ELSE**, **THEN**, **DO**, **LOOP**, and **+LOOP**.

Adverbs tell how, when or where. They modify adjectives, verbs and other adverbs. They are something like adjectives and something like verbs. In Forth they are words with stack diagram

(n1,...n*i*,n[*i*+1],...n*k* -- n1,...n*i*)

They take items off the stack, but leave some unchanged.

Thus, all eight parts of speech are found in Forth. Other grammatical features are also present.

A phrase is a sequence of words which could be used as a colon definition. A phrase has a stack diagram which tells what kind of phrase it is. Thus, if the stack diagram is (-- n), it is a noun phrase.

A clause is a verb phrase or an adverb phrase. Dependent clauses are adverb phrases; independent clauses are verb phrases.

In English, just the first letter of the name of a language is capitalized (*e.g.* English, French, Fortran, Forth). Thus, **FORTH** is a Forth word. The name of the compiler will depend upon your system, but will probably be "forth" or "FORTH."

When I first thought of these correspondences, I thought they were amusing but not very useful. A year later, I think they have a definite value. They suggest a way of choosing names for words, a convenient classification when describing a word and a rationale for punctuating phrases in definitions. A clause, for instance, should be separated by end-of-line or by extra spacing.

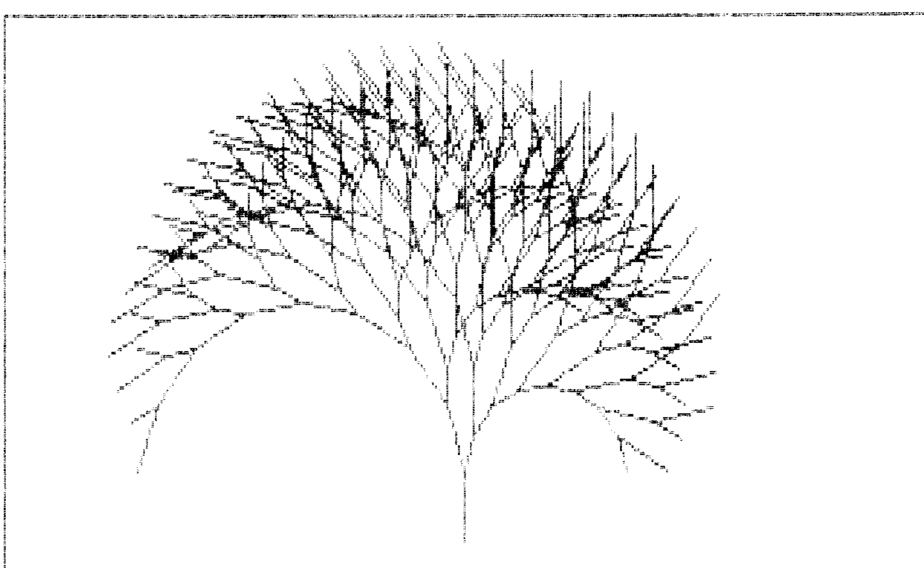
Vectored Execution and Recursion

Roy W. Sommers
Pennsville, New Jersey

```
0 VARIABLE 'LBR     ( Used to store PFA of (LBR) )
0 VARIABLE 'RBR     ( Used to store PFA of (RBR) )
8 VARIABLE LEVEL    ( Value of calling level )
20 VARIABLE ANGLE   ( Angle between stems )
: RETURN R> DROP ;     ( Returns execution to calling level )
: LBR 'LBR @ CFA EXECUTE ;   ( Define LBR in terms of 'LBR )
: RBR 'RBR @ CFA EXECUTE ;   ( Define RBR in terms of 'RBR )
: NODE
    LEVEL @ 1 < IF RETURN THEN
    -1 LEVEL +! ANGLE @ LEFT ( Adjust level and turn )
    LBR         ( Draw left branch )
    ANGLE @ 2* RIGHT           ( Turn )
    RBR         ( Draw right branch )
    ANGLE @ LEFT 1 LEVEL +! ; ( Reset )
: (LBR)
    DUP 2* PENDOWN FORWARD     ( Draw 2x stem )
    NODE        ( Do next level )
    DUP 2* PENUP BACKWARD ;    ( Reset Cursor )
: (RBR)
    DUP PENDOWN FORWARD        ( DRAW 1x stem )
    NODE        ( Do next level )
    DUP PENUP BACKWARD ;       ( Reset Cursor )

' (LBR) 'LBR !     ( Store PFA of (LBR) in 'LBR )
' (RBR) 'RBR !     ( Store PFA of (RBR) in 'RBR )

: SETUP 8 LEVEL ! 10 LBR DROP ; ( Type SETUP to draw tree )
```



Previous articles on recursion (*e.g.* Vol. IV, No. 2) utilized the words **MYSELF** and **RETURN** to effect recursion. However, in cases where the procedures call each other, this approach does not work. Recursion can still be accomplished, though, by using vectored execution.

This type of recursion can be illustrated in terms of the binary tree described in *Turtle Geometry* (Abelson and diSessa, page 83). The reference program was written in a form of LOGO and used separate procedures to draw left and right stems of different lengths, and a third procedure to control depth of recursion and cursor positioning. Each procedure called one or more of the other procedures.

An analogous program can be written in Forth by using vectored execution (see accompanying code and illustration). Since **NODE** contains **LBR** and **RBR**, these words must be defined before **NODE** is loaded. However, if they were written in the form **(LBR)** and **(RBR)**, they could not be loaded before **NODE** since they contain **NODE**. Vectored execution solves this problem by defining **LBR** and **RBR** in terms of the variables **'LBR** and **'RBR** which, at the time of execution of **NODE**, contain the PFAs of **(LBR)** and **(RBR)**.

Apple Forth á la Modem

```
SCR # 120
( TERMINAL MODEM ROUTINE  -RDA 8/06/83 )

HEX
COB5 CONSTANT CTRL2
COB6 CONSTANT CTRL1   ( WRITING )
COB6 CONSTANT STATUS  ( READING )
COB7 CONSTANT DATA
 3CO CONSTANT BYTES/SCR ( APPLE SCREEN)

: IN?MOD  ( -- BITO ) STATUS C@ 1 AND ;
: OUT?MOD ( -- BIT1 ) STATUS C@ 2 AND ;

: OUT.MODEM  ( CHAR -- )
  BEGIN OUT?MOD UNTIL DATA C! ;

: INIT.MODEM ( 1=ORIG 0=ANS  -- )
  3 CTRL1 C! 11 CTRL1 C!
  IF 8F ELSE 8B THEN CTRL2 C! ;

: GET.MODEM ( -- CHAR )
  BEGIN IN?MOD UNTIL DATA C@ ;

 DECIMAL
     ;S
OK
```

```
SCR # 121
( TERMINAL ROUTINE FOR MODEM PAGE 2 )

 HEX
: TRANS#   ( FROM.LOCAL# TO.REMOTE# -- )
  CR 2 ( CONTROL B )
  OUT.MODEM OUT.MODEM BLOCK
  5000 0 DO LOOP BYTES/BLK 0
  DO DUP I + C@ DUP EMIT OUT.MODEM
  LOOP DROP ;

: RTRANS# ( F.LOCAL# T.REMOTE #BLKS -- )
  0 DO OVER I + OVER I + TRANS#
       LOOP 2DROP ;

: RECV GET.MODEM BLOCK BYTES/BLK 0
  DO GET.MODEM DUP EMIT OVER I + C!
  LOOP DROP UPDATE ;


  DECIMAL   ;S

```

(Listings Continued)

R. Dudley Ackerman
San Francisco, California

These words will allow Forth users with Hayes Micromodems and Apples to send screens back and forth. Minor modifications will allow use on other systems.

Execute **MODEM** after connect is made. A one should be on the originator's stack, a zero on the stack of the computer in answer mode. Both parties should be able to see entries from both keyboards.

When the modem program sees a control-B from its keyboard, it inputs a line of Forth. By executing **RTRANS#**, a set of screens can be sent from one system to the other. Notice the stack setup for **RTRANS#** : source screen number, destination screen number, and number of screens.

TRANS# sends a control-B out and a screen number, then waits to give the receiver time to get the screen into its buffer. When the modem program gets a control-B across the line, it executes the Forth word to receive a screen.

@KEY is Apple specific and gets a character from the keyboard. Hex eleven in **INIT.MODEM** sets the number of bits per character to eight, with no parity, one start bit, and two stop bits for the Hayes modem. Hex three initializes the Hayes modem.

The program will also function as a simple terminal program. One desirable feature would be a way to capture text into free memory, then save the text to disk.

```
SCR # 122
( MODEM                          RDA  8/06/83)

 HEX
: @KEY ( -- KEY )
  C000 C@ C010 C@ DROP 7F AND ;

: MODEM INIT.MODEM
  BEGIN ?KEY DUP
   IF DROP @KEY DUP 2 =  ( CTRL 'B' )
       IF DROP QUERY INTERPRET
       ELSE DUP EMIT DUP OUT.MODEM THEN
   THEN 51 -  ( 'Q'=QUIT )
  WHILE IN?MOD
    IF DATA C@ DUP 2 =  ( CTRL 'B' )
        IF DROP RECV ELSE EMIT THEN
    THEN
  REPEAT ;


  DECIMAL

  ;S


OK
```

End Listing

Forth–83 Loop Structure

Bill Stoddart
Middlesbrough, England

The story so far...

Bob Berkey has suggested a new loop structure capable of covering a full 64K range, of handling positive or negative increments, or even increments which switch sign. The internal implementation is based on the fact that an overflow condition occurs when a sixteen-bit addition or subtraction traverses the boundary between 7FFF and 8000 hex in either direction. By using 8000 hex as a universal loop limit and performing a corresponding transformation on the loop index, we can test for completion of the loop by checking whether adding the increment to the transformed index causes an overflow.

The new loop has been accepted into the 83-Standard, but requires careful thought if its advantages in terms of generality and speed are to be accompanied by simplicity of implementation.

The main complication of the new loop is in the implementation of **LEAVE**, which has traditionally equated the loop limit and index, forcing termination on the next occurrence of **LOOP** or **+LOOP**. This technique is no longer available, as there is no longer an explicit loop limit, and there is no value the index can be set to which will ensure termination for both positive and negative increments. Setting the limit to 7FFF hex will ensure termination for **LOOP** and **+LOOP** with a positive increment, since adding a positive value to 7FFF will always cause an overflow. However, there will be no overflow if the index is decremented by **+LOOP**.

With this in mind, the 83-Standard specifies that **LEAVE** should straight away transfer execution to just beyond the end of the loop structure. Various ways of achieving this have been suggested. Bob Berkey's original suggestion was that the runtime operation compiled by **DO** should push an exit address onto the return stack. This im-

```
180
( Nucleus )

HEX

CODE (DO)
   AX POP ( initial index )   CX POP ( limit )   AX CX CMP
   0= IF ( bypass null loop )  LODS   AH AH SUB   AX SI ADD
   ELSE   BP SP XCHG   8000 # DX MOV   CX DX SUB   DX PUSH ( x )
     AX DX ADD   DX PUSH ( i )   SP BP XCHG   SI INC
   THEN   NEXT   END-CODE

DECIMAL
```

```
181
( Nucleus )

CODE (LOOP)   BP@ WORD INC   OFL IF   4 # BP ADD   SI INC
   ELSE   LODS   AH AH XOR   AX SI SUB   THEN   NEXT   END-CODE

CODE (+LOOP)
   AX POP   AX BP@ ADD   OFL IF   4 # BP ADD   SI INC
   ELSE   LODS   AH AH SUB   AX SI SUB   THEN   NEXT   END-CODE
```

```
182
( Nucleus )

CODE I ( -- n   leave loop index )
   BP@ AX MOV   2 DISP8 BP@ AX SUB   AX PUSH   NEXT   END-CODE

CODE J ( -- n   leave outer loop index )
   4 DISP8 BP@ AX MOV   6 DISP8 BP@ AX SUB   AX PUSH   NEXT
   END-CODE

CODE (LEAVE)
   4 # BP ADD   LODS   AH AH SUB   AX SI SUB
   LODS   AX SI ADD   NEXT   END-CODE
```

```
183
( System word set, high level )

: <MARK ( -- addr )   HERE  ;

: <RESOLVE ( addr -- )
   HERE SWAP - 1+   C,  ;

: >MARK ( -- addr )   HERE   0 C, ;

: >RESOLVE ( addr -- )
   HERE   OVER - 1-   SWAP C!  ;
```

```
184
( Program structures )

VARIABLE CLUE

: DO ( -- dest )
   CLUE @   COMPILE (DO)   >MARK ( forward branch past loop )
   DUP CLUE !   <MARK ( backward branch from loop )  ; IMMEDIATE

: LOOP ( dest -- )
   COMPILE  (LOOP)   <RESOLVE   >RESOLVE   CLUE !  ;  IMMEDIATE

: +LOOP ( dest -- )
   COMPILE (+LOOP)   <RESOLVE   >RESOLVE   CLUE !  ;  IMMEDIATE

: LEAVE ( addr1 addr2 -- addr1 addr2 )
   COMPILE (LEAVE)   CLUE @   <RESOLVE  ;  IMMEDIATE
```

End Listing

```
... DO   TEST   IF CONNECT LEAVE THEN   LOOP ...

This compiles as:
```

(DO)	14	TEST	?>BRANCH	5	CONNECT	(LEAVE)	11	(LOOP)	13

Figure One

plies a runtime penalty whether or not **LEAVE** is included in a loop, and subsequent suggestions from Klaus Schleisiek and Bill Ragsdale have avoided any runtime penalty by using **LOOP** or **+LOOP** to resolve forward branch addresses left by an **IMMEDIATE** version of **LEAVE**.

Now read on...

The implementation presented here runs loops at maximum speed and also avoids any complexity in the compile-time behavior of **LOOP** or **+LOOP**. These words do not need to know about the existence of **LEAVE**. There is a minimal runtime penalty when the runtime operator for **LEAVE** is executed. The essential idea of the implementation is that the runtime operator **(LEAVE)** compiled by **LEAVE** calculates its continuation address by locating an offset which follows the runtime operator **(DO)** compiled by **DO**.

Consider figure one. The runtime operators **(DO)**, **(LOOP)** and **(LEAVE)** are followed by unsigned eight-bit displacements. Thus, **(DO)** has an offset to beyond the loop, and **(LOOP)** has an offset back to the start of the loop.

The offset that follows **(LEAVE)** is back to the location following **(DO)**. **(LEAVE)** performs its function by:
a) subtracting its offset from IP, leaving IP pointing to the byte that follows **(DO)**, and
b) adding the offset that follows **(DO)** to IP, leaving IP pointing just beyond the loop.

At compile time, **DO** saves the loop start address in the variable **CLUE**, having first saved the previous value of **CLUE** on the parameter stack. **LEAVE** uses the contents of **CLUE** to calculate the offset back to the start of the loop. **LOOP** or **+LOOP** will restore the previous value of **CLUE**, which will either be its initial value or the start address of an outer loop.

The code presented here is without compiler protection, but setting the initial value of **CLUE** to zero provides a simple test for being inside a loop.

The existence of an offset following **(DO)** means that including a test for a null loop is very simple and efficient. On grounds of functional correctness, too, I think this should be included in the standard, and the test is included here.

The code definitions of the runtime operators are given for an 8086 implementation. For those who wish to ponder the details, SI is a sixteen-bit index register used as Forth's IP. Next is post incrementing, so that during execution of a code-level word, IP points to the following byte.

BP is a sixteen-bit index register used as Forth's RP. The return stack grows toward low memory, so that

```
4 # BP ADD
```

drops two items from the return stack.

AX is a sixteen-bit accumulator with low byte AL and high byte AH. The @ symbol is used in the assembler to denote indirection, so

```
AX BP@ ADD
```

will add AX to the contents of the location indicated by BP, which is the top of the return stack. The instruction

```
2 DISP8 BP@ AX ADD
```

adds the second return stack item to AX.

LODS is a one-byte instruction equivalent to

```
SI@ AL MOV   SI INC
```

The offsets compiled by **DO**, **LEAVE**, **LOOP** and **+LOOP** are calculated using the 83-Standard system words **>MARK**, **>RESOLVE**, **<MARK** and **<RESOLVE**. The definitions of these words are included for completeness. They are specific to a system using eight-bit unsigned offsets.

The definitions of **DO**, **LEAVE**, **LOOP** and **+LOOP** will be usable on any system, since they use the system words to hide details of the underlying implementation.

Appendix: DO...LOOP Algebra

It is fascinating that a discovery such as the new loop can be made after so many programmers and compiler writers have been implementing loops for so many years. One reason why this type of thing is tricky is that sixteen-bit integer arithmetic is really two arithmetic systems combined, signed and unsigned, and in using the overflow flag to test for loop termination we are using a facility from signed arithmetic to test for the completion of an unsigned operation. In a hardware realization of a Forth machine, of course, the internalized limit could be zero, with a "loop" flag which comes up when a sixteen-bit arithmetic operation crosses zero.

Let L and I represent the loop limit and index. We define the range R of the loop as $R = L - I$.

We transform L and I to internalized values l and i, where in most systems $l = 8000$ hex.

This transformation does not affect the range of the loop, so we can write $R = l - i$, and thus $i = l - L + I$.

As the loop executes, there is a constant difference between i and I which can be expressed as $I = i - x$, which by eliminating i gives $x = l - L$.

The values of x and i are calculated by the runtime operator for **DO** and are pushed onto the return stack. (See comments in **(DO)**.) The Forth word **I** calculates the loop index using $I = i - x$.

Multi-Tasking, Part I

Henry Laxen
Berkeley, California

Multi-tasking has long been one of the biggest benefits of Forth and one of its most closely guarded secrets. The fact that even crippled processors like the 8080 can be made to run four or five tasks simultaneously with little performance degradation is a testament to the efficiency both of Forth itself and of the techniques involved in implementing a multi-tasking kernel. It is time to reveal the techniques used in most Forth multi-tasking systems and to allow the user to benefit from the power such knowledge can bestow.

Now for the disclaimers. First, I am only going to discuss multi-tasking, and not multi-user, Forth systems. The difference is that in a multi-tasking Forth there is but one terminal attached to the system, hence only one person at a time is interpreting or compiling. This is considerably simpler than a multi-user Forth system with several terminals (each perhaps with its own unique characteristics), all interpreting and compiling at the same time. In our multi-tasking system, the user will be able to have several tasks running simultaneously, perhaps communicating with each other and with the terminal, but I will not get into the subtleties associated with turning it into a true multi-user environment. The second disclaimer is that in order to get some efficiency out of the system, the techniques used to implement multi-tasking are generally very machine dependent. Since my machine is an 8080, and it is me writing this article, you will either have to bear with me or ignore the article. The choice is yours.

This first installment will deal with the low-level mechanism involved in task switching, and the structures that must be in place in Forth in order to make multi-tasking possible. The second article on this subject will talk about creating and manipulating tasks. While the code I present is oriented toward an 8080, I will describe its function. You should be able to translate it into code for your processor without much pain.

Now then, let me first describe the philosophy behind multi-tasking in Forth. Unlike traditional multi-tasking systems, which interrupt the currently running task at a completely arbitrary time and initiate another task unbeknownst to the first one, Forth requires tasks to cooperate. While each task does not know details about the other tasks, it must at least be aware of them, or else the system will revert to a single-task existence. Each task must explicitly give up control of the CPU at certain points while it is running. The Forth kernel does so whenever it is about to perform an I/O operation, such as reading or writing to the terminal or mass storage device. If the user creates a task that does no I/O of its own, then he must explicitly give up control or else as soon as that task is activated, it will take total control of the machine. Each of the two approaches mentioned above has merit, and I will briefly discuss the good and bad points of each.

The main advantage of traditional multi-tasking systems is that the programmer does not need to be aware of their existence. As far as he is concerned, the program he writes is just running slower. He does not need to modify it in any way from a single-user environment. There is, of course, a cost associated with this benefit, and that is one of performance. Since the operating system must absolutely guarantee that the state of the system is undisturbed between one running of the task and another, an extremely complex process usually is required to save and restore a task. Since the task is unaware that it is being removed from control, the operating system may grab it at a particularly inopportune moment, and the amount of information that must be saved and restored can be staggering. This is why performance on such systems typically degrades rather dramatically as soon as several tasks are running concurrently. The main advantage of the Forth approach to multi-tasking is that the overhead of task switching is extremely small.

Thus, many tasks can be run simultaneously with little performance degradation. The disadvantage is that an additional burden is placed on the programmer. He must follow some rules that apply in multi-tasking systems, as well as, perhaps, modify his code to take advantage of multi-tasking. My conclusion from this synopsis is that traditional multi-tasking is great on very large systems where tens or hundreds of tasks are running simultaneously and the computer hardware helps you. On microcomputers and small systems, the traditional approach simply does not apply, and the benefits of

```
VARIABLE UP ( Points to the currently active user base address )
: USER   CREATE ,   DOES>   @ UP @ +   ;
```

Figure One

```
VARIABLE #USER   ( Holds the size of the user area )   0 #USER !
VOCABULARY USER   USER DEFINITIONS
: CREATE   CREATE #USER @ ,   DOES>   @ UP @ +   ;
: ALLOT    #USER +!   ;
: VARIABLE CREATE   2 ALLOT   ;
FORTH DEFINITIONS
```

Figure Two

the Forth approach greatly outweigh the drawbacks.

The basic mechanism Forth uses is simply to define an ordinary Forth word, usually called **PAUSE**, which relinquishes control of the CPU from the currently running task and gives it to the next task that is ready to run. **PAUSE** takes nothing from the stack and returns nothing, and disturbs the system in a well-defined way of which the user must be aware. What **PAUSE** actually does is to examine a linked list of tasks that may or may not be ready to run. The first ready task it finds is given control of the CPU and is allowed to run until it executes a **PAUSE** of its own. The linked list is circular, so eventually we will get back to the first task in the list and run it again, with execution continuing immediately after the **PAUSE** word. By agreement, tasks shall not disturb the state of the system except with regard to block buffers. Thus, each task may not assume that the buffer it is using is still present after a **PAUSE** has been executed. This minor restriction greatly simplifies the job of saving and restoring the state of the system between task activations.

Wait a minute, you say, what about the many system variables that tasks may use while they are running. For example, if a background task is doing print spooling while you are editing a screen, both tasks are accessing variables such as **OUT**, **BASE**, **HLD**, etc. Things would get very confusing if each task could change these and affect other tasks. Fortunately, there is an extremely elegant way to prevent this which has traditionally been known as **USER VARIABLE** in Forth. The idea is simple, namely, just group together those variables which each task must have to itself. These variables become offsets from some base address. When these variables are executed, they must add their offset to the base address of the current task. Thus, to switch tasks one need only change the base address from whence these variables originate, instead of copying the values themselves to some safe area. The portion of memory reserved for these variables is called the **USER** area. There are many different ways of implementing this concept, and I would like to present a new one here which I believe has great merit. Traditionally, **USER** was a defining word which took as an argument an offset from the base address and assigned a name to that offset. At runtime, the offset was added to the base of the current user area, which was contained in a regular variable, and that address was placed onto the parameter stack. This is simple, but has some disadvantages. It is difficult to insert a new **USER** variable into the middle of existing ones with this implementation. It also forces the user to be aware and do arithmetic in order to maintain the user area. The old implementation was as shown in figure one.

```
CODE PAUSE    (S -- )
   IP PUSH   ( Push the current interpreter pointer onto stack )
   RP LHLD   H PUSH    ( and the current return stack pointer )
   0 H LXI   SP DAD   XCHG    ( Stack pointer now in DE )
   UP LHLD   ( Points to TOS, which is first entry )
   E M MOV   H INX   D M MOV   H INX    ( Move stack pointer to TOS )
   H INX   PCHL    ( Jump to next task )    C;
```

Pause on the 8080
Figure Three

```
CODE RESTART    (S -- )
   ( Since a RST instruction has just been executed, the address
     UP + 3 is now on the stack )
-3 H LXI   D POP   D DAD   UP SHLD    ( Set up new USER area )
M E MOV   H INX   M D MOV   XCHG   SPHL   ( Restore parameter stack )
H POP   RP SHLD    ( Restore Return Stack Pointer )
IP POP    ( Restore the IP )    NEXT JMP    C;
```

Figure Four

A much more flexible approach is to make **USER** a vocabulary, and redefine those words which may be needed on a task level. Consider the implementation in figure two.

Now you need no longer keep track of where each variable is going and how much space has been used. Also, arrays are much easier to create, and I think it reads much more nicely. With the old approach, you would have to say **34 USER BASE** to define a user variable called "base," and you must know that location 34 is available for use. With the new scheme, you simply type **USER VARIABLE BASE**, which reads very nicely indeed.

Now then, suppose we have such things as **USER** variables, regardless of exactly how they were defined. In particular, I will need three such variables as follows:

USER VARIABLE TOS (Holds top of stack when switching tasks.)

USER VARIABLE ENTRY (Contains machine code and task status.)

USER VARIABLE LINK (Points to next task in a circular list.)

Let's examine the role of each of these a little more closely. **TOS** is simply going to hold the value of the top of the parameter stack for this task, when it gives up control of the CPU to the next task. Since each task must have its own local stack in order to do just about anything, this value must be saved and restored between successive activations of a task. **ENTRY** in our implementation will contain machine code that will either jump to the next task in the list if the current one is not ready to run yet, or it will jump to some activation code that will bring this task to life once again. Finally, **LINK** points to the **ENTRY** field of the next task in the circular list. The only tricky part of this is how to fit the code that decides whether or not to activate this task and either continue or restore all of the task's parameters, in the two bytes reserved for **ENTRY**. It just so happens that, on the 8080, two bytes is more than enough and, in fact, one would suffice.

The 8080 has several one-byte instructions called RST instructions. When these are executed, they push the value of the program counter on the stack and jump to a specified location in low memory. Thus, the trick on the 8080 is to put either an RST or a JMP into the **ENTRY** point. An RST instruction will cause this task to be activated, while a JMP instruction will jump immediately to the **ENTRY** point of the next task in the list. Remember that the contents of **LINK** point to the **ENTRY** point of the next task in the list. So to make a task active, an RST instruction is placed into **ENTRY** while to deactivate a task an NOP instruction is placed into **ENTRY**. The JMP instruction is always present in **ENTRY + 1**. This is wasteful, I know, but what the hell. Now then, all we have to do is understand what exactly happens when we do a **PAUSE** and a task activation. Let's first look at what **PAUSE** does on the 8080. (See figure three.)

PAUSE is in charge of saving the current task's status and jumping to the next task in the circular list. Notice how little information needs to be saved during a task switch. Only the current value of the IP, the return stack depth, and the parameter stack depth is saved. Note that the IP and the return stack depth have been pushed onto the parameter stack, so it will be the duty of the **RESTART** word to pop these off so that the stack depth is unchanged. Now let's take a look at **RESTART** in figure four, which must restart a task where it left off, namely just after executing a **PAUSE**.

Remember that the RST instruction is a one-byte call to a fixed address. Thus, it pushes the address of the current user area plus three onto the current stack. This information is used to restore the user area for the task that is now being restarted. Once the base of the user area is computed, the parameter stack is restored and then the return stack and the interpretive pointer. Thus, **RESTART** has undone what was done by **PAUSE**, and resumed execution with the word following **PAUSE**, as though nothing has happened.

I hope this has shed a little light on what goes on in a multi-tasking system. Next time, we will explore how to create and manipulate tasks, now that we understand the task-switching mechanisms involved. Until then, good luck, and may the Forth be with you.

 The author is Chief Software Engineer for Universal Research, 150 North Hill Drive #10, Brisbane, CA 94005, specializing in the development of portable computers.

Letters *(Continued from page 4)*

fact, any reference to an operating system at all (except for terminal interfacing words like **EMIT**), as far as the standard is concerned. That is most definitely not to say that the screen system is not a good one. In fact, it might be the best system for particular applications. But I think Forth should emulate the attitude of C and Modula II in that respect. Separate the design and implementation of the operating system from the language standard.

The next issue on my mind is that of strings. Possibly one of the FIG study groups is taking care of my concerns; but it would make me feel more comfortable to see some debate, at the level of fundamentals, in *Forth Dimensions* before things get too far along.

BASIC has taught us that good string handling is one of the very important components of interactiveness. The secret of (Microsoft) BASIC's friendliness with strings is that you never have to specify the length of a string, and the secret of that is BASIC's string-space garbage collector. Here is an opportunity for Forth, because, however they are implemented, Forth's string variables are likely to be dispersed randomly in the code, rather than being assigned to a well-defined memory region; and that makes garbage collection impossible without both forward and backward links between string variables and string values in string space. This is extra memory overhead, compared to BASIC, but the advantage is that it makes very fast garbage collection possible.

String handling is very important, and Microsoft BASIC has shown us that it can be done extremely well in small-scale systems. We should accept nothing less for Forth.

Finally, let me comment on the editor's reply to a letter in *Forth Dimensions* (Vol. IV, No. 6, page 25), pointing out a design bug in a FIG release. The response was to the effect that the bugs were known, and had been fixed in the implementations of various vendors, who have more resources for that kind of maintenance than FIG.

I have to chide you a little for that. We all appreciate the tremendous generosity of the people behind FIG, who put their public-domain philosophies where their mouths are, at the cost of that most valuable commodity, personal time. But the situation is surely analogous to that of publishing research in scientific journals. As a theoretical physicist, it is my professional and ethical responsibility to publish an erratum if I become aware of a significant mistake in any of my published work. I think the same should apply to FIG publications. If you know of a bug, you should publish an erratum.

I hope that, in the interest of pruning my remarks to a few, I have not conveyed an overly critical impression. I really get a sense, from two years' worth of *Forth Dimensions*, that the community is making solid advances. Although not a Forth fanatic, let me

(Continued on next page)

```
Screen # 32
( Calander Development, Screen 1 of 3 )
: CTABLE <BUILDS 0 DO C, LOOP DOES> + C@ ;
  31 30 31 30 31 31 30 31 30 31 29 31 0
  31 30 31 30 31 31 30 31 30 31 28 31 0
26 CTABLE DAYS-IN-MONTH ( month -- days in month )

: IS-LEAP-YEAR? ( year -- flag )
  DUP DUP 400 MOD 0=          ( if year divisable by 400 )
  SWAP    100 MOD 0= NOT OR   (    or not divisable by 100 )
  SWAP      4 MOD 0= AND ;    (  and divisable by 4 then leap )





-->
```

```
Screen # 33
( Calander Development, Screen 2 of 3 )
: DAY-OF-YEAR ( day, month, year -- day of year )
  IS-LEAP-YEAR?
  IF 13 + 14              ( if leap year convert offsets )
  ELSE 1 ENDIF            ( else start at month 1 )
  2DUP = IF 2DROP         ( if month is january, return day )
         ELSE DO I DAYS-IN-MONTH + LOOP ( else cal day of year )
         ENDIF ;
: DAY-OF-WEEK ( day, month, year -- day of week, sunday is 0 )
  DUP >R DAY-OF-YEAR 0   ( get current day of year, current day )
  R> 1900 DO              ( start at year 1900)
    I IS-LEAP-YEAR? IF 366 + ELSE 365 + ENDIF
  LOOP +                  ( add in day of this year )
  7 MOD ;                 ( mod by 7 for day of week )
  ( Note: works from year 1901 to 2100 )
-->
```

```
Screen # 34
( Calander Development, Screen 3 of 3 )
: CALANDER ( month, year -- ) ( print months calander )
  2DUP SWAP . ." - " . CR
 ."  Sun Mon Tue Wen Thr Fri Sat" CR
 2DUP 1 ROT ROT DAY-OF-WEEK    ( find day of 1st day of month )
 DUP 2* 2* SPACES              ( space over in output )
 ROT ROT                       ( stack: dayofweek, month, year )
 IS-LEAP-YEAR? IF 13 + ENDIF DAYS-IN-MONTH ( get days in month )
 1+ 1 DO
   I 4 .R                      ( out day )
   1+ DUP 7 = IF CR ENDIF 7 MOD ( if sat then cr )
 LOOP 0= NOT IF CR ENDIF CR ;  ( output ending crs )
: CALANDER-YEAR ( year -- ) ( print calander for whole year )
  13 1 DO DUP I SWAP CALANDER LOOP DROP ;


```

```
7 1983 CALANDER
7 - 1983
 Sun Mon Tue Wen Thr Fri Sat
                       1   2
   3   4   5   6   7   8   9
  10  11  12  13  14  15  16
  17  18  19  20  21  22  23
  24  25  26  27  28  29  30
  31
```

end by saying that I find FIG a uniquely valuable enterprise.

David N. Williams
1238 Westport
Ann Arbor, MI 48103

Code for All Seasons

Dear FIG,

At last year's Forth convention in San Jose, one speaker mentioned that functions to print out the calendar would be useful. Though I am not an expert in Forth yet, these functions seemed simple, and I have developed the code shown in the accompanying screens. I am using a Forth developed from the FIG model; it should be simple to convert these functions to Forth-79.

I would personally like to see some routines that would display a comment associated with a Forth word. I find it very awkward to look up uncommon definitions each time I need to use one. Currently, I am just not familiar enough with Forth to implement this. Does anyone have some ideas?

Thank you very much,

Jesse Jay Wright
164 N. Oak Knoll #8
Pasadena, CA 91101

FIG Aficionado

Dear Editor,

Re: the contents of *Forth Dimensions*, you find me delighted to see true FIG-Forth screens reappearing. After all, FIG-Forth is our group's own version of Forth, and though the "standard" (unfortunately, I think) gets changed every four years or so (this is bound to scare off many a potential commercial user worried about software maintenance — a standard that changes is a contradiction in terms!), FIG-Forth escaped unscathed. In contrast, a Forth-79 screen written only one year ago won't necessarily load on a Forth-83 system. Try to explain this to a business user who just got coaxed into spending a few thousand bucks on a Forth-83 system with promises of a wealth of ready-made software at large, and who discovered that it will require considerable amounts of rewriting before he can use it. The same applies to Forth-79 users who won't be able to load all these '83 screens which, no doubt, will appear before long. If new "standards" keep popping up like this, there will soon be as many Forth dialects as there are BASICs, and the intended source portability will be nothing but a hollow phrase.

A while ago, I obtained a Z-80 FIG-Forth listing from Dennis Wilson of Aristotelian Logicians. To do better justice to the Z-80 bit, I re-worked parts of it. I got lots of fun out of re-writing code definitions, ending up with fewer bytes used and a faster execution time to boot. As a result, the Primes benchmark (*Forth Dimensions*, Vol. II, No. 4) executes 1000 primes in fifty-four seconds, and the *BYTE* test (September 1981) runs in just eight seconds. I'd very much like to hear from anyone who can do better than this on a 4Mz, Z-80 system. I will gladly supply the source text to any interested person who sends a 5 1/4'' double-sided, single-density diskette (or $5) and return postage.

Yours Forthfully,

Edmund Ramm
Postfach 38D-2358
Kaltenkirchen, West Germany

New Product Announcements

Forth Dimensions *welcomes press releases and product announcements, as well as reader letters regarding product performance. Addresses of the distributors and manufacturers mentioned in this column may be found in the Vendors List.*

Perkel Software Systems has announced that its **Marx Forth v1.4** is now a public-domain product. Included is its target compiler system which allows applications to be run as stand-alone machine code files that don't require a Forth system to run. Source listings and forty-page manual cost $35, disk version $150; for NorthStar, CP/M, Atari, Radio Shack. Those interested in developments related to a "universal compiler" with the ability to compile code from Forth, Pascal, C and BASIC may write to Perkel Software Systems.

The **Pocket Guide to Forth** lists Forth words in ASCII order, along with definitions and stack diagrams. Gives FIG-Forth and Forth-79 correlations. Available for $7.00 from Mountain View Press ($7.25 elsewhere).

Forth Inc. has released a schedule of **Forth classes** for the period from November 1983 to February 1984. Included are an intensive introduction covering vectored execution, array handling, sealed vocabularies, data typing, data formatting and management; advanced instruction detailing multi-tasking, serial device drivers, interrupt routines, **BLOCK** device drivers and target compile applications culminating in a running target image; and data-base design concentrating on the techniques needed to store data on disk and to design reports, user interface and security, indexing methods, and data description and access tools. Courses are five days in length, consist of both lecture and hands-on learning, and cost between $750 and $950 for individual enrollment.

Several printed listings which include the **83-Standard** are available from MicroMotion for $15 each. Also available are 6502, 8080 and 8086 source listings which support both the new standard and the 83 model by Laxen and Perry. Call or write them for a free **83-Standard Programming Reference Card**.

Ziggurat Software announced applications for **HES' FIG-Forth for the VIC-20**. Included are additional FIG-Forth words not in the HES implementation, printer utilities for the VIC printer, a case statement, arrays and strings, and disk utilities. Available on cassette or disk.

Innovatia Laboratories offers three products. FMS is a **text formatter** which permits storage of text or direct output (typewriter-like) to printer, and permits access to all upper- and lower-case Greek letters and forty-six math and special symbols. FLH provides **LISP-like list-handling** in Forth and a small expert system written in FLH. FWG, a firmware generator, creates **ROMable Forth code**, with or without headers, for target compilers.

Forth for the **Texas Instruments 99/4A** can be purchased from Wycove Instruments Ltd. It requires at least 32K of additional memory and one of the following: Editor/Assembler, Mini Memory or Extended BASIC. A cassette version is available. Sprites, sound, floating-point arithmetic and peripheral access are supported. The $50 price includes a short introduction to Forth, complete description of the included Forth words, hardware notes and a sample game. Beginners are referred to *Starting Forth* by Leo Brodie.

Chapter News

John D. Hall
Oakland, California

We have three new chapters! They are:

Fox Valley FIG Chapter
Batavia, Illinois

Philadelphia Area FIG Chapter
Philadelphia, Pennsylvania

Houston FIG Chapter
Houston, Texas

The following areas do not have FIG chapters, even though there are sufficient FIG members to form them. I know that many members in these areas are interested, but someone will have to make an effort!

Tucson, AZ	Atlantic City, NJ
San Jose, CA	Corvallis, OR
Gainesville, FL	Eugene, OR
Tampa, FL	Columbus, OH
Orlando, FL	Cincinnati, OH
Huntsville, FL	Norfolk, VA
Melbourne, FL	Roanoke, VA
Atlanta, GA	Richland, WA
Indianapolis, IN	Madison, WI
Baton Rouge, LA	Milwaukee, WI
Raleigh/Durham, NC	

Australia Chapter

At the meeting on July 1, after a question and answer session, the discussion was devoted to screen transfer, with some heavyweight thinking about the ISO seven-layer model. On August 5, the same topic was featured, and at the end of the meeting, several people decided to actually do something about writing code and making cables to actually transfer some data.

Los Angeles Chapter

There was a talk on August 27 about the current state and future trends of software and hardware development for personal and business computers in Japan. Martin Tracy discussed techniques and implementations for ROM-able Forth systems. Barry Cole explained how he implemented a quick booting Forth on a portable computer.On September 24, Nathaniel Grossman described his method for implementing nine decimal place, binary logarithms in fixed-point Forth. John Hall, FIG Chapter Coordinator, gave a report on the activities of other FIG chapters. There was some discussion on the way to increase communication between chapters. Later, the group talked about implementation of Forth-83, which may lead to a model. Martin Tracy demonstrated and described the MicroMotion implementation of Forth-83. Greg Stevenson discussed a method of speeding dictionary searches independent of the existing method. At the October 22 meeting, Nathaniel Grossman spoke on finding 16-bit square, cube, *etc.* roots by Newton's method, in fixed-point Forth. Bob Jaffray presented a simple method to provide execution security by testing for valid CFAs as a patch to **NEXT**. Steven Lewis talked about **VERIFY**, a word to document effects of the execution of a Forth word. Jim White described an implementation of LISP in Forth. Ken Inouye demonstrated Forth on the new Sharp 16-bit CPU computer. There were also reports from members who attended the National Forth Convention. About thirty-five people attended this chapter meeting and enjoyed it very much.

Iowa Chapter (in formation)

August 23, the group saw a presentation of a simple but fast LIFE implementation on the Commodore 64 (using C64-Forth from Performance Micro Products) by Robert Benedict, Assistant Professor of Mechanical Engineering. One generation in Forth was 1.6 seconds vs. 126 seconds in BASIC. Another demonstration by Scott Evans, an electrical engineering student, showed some Forth utilities developed for the Commodore 64, used as a controller to move a hydrophone in a plane, for data acquisition.

Eugene Johnson, a professor of mathematics, presented a series of words on September 27 to do matrix operations. He uses these in his courses on linear algebra. Michael Ham showed some words that simply ignore invalid input. These are useful when the range of valid input is obvious, as from a menu. The effect for the user is that the only keys that work are those for valid input.

Orange County Chapter

Roy Martens paid a brief visit at the August 24 meeting, and discussed what was happening with the Northern California Chapter, and the state of the Forth-83 standard. Lee Jordan presented a paper on a 6502 disassembler. Lee is a beginning Forth programmer and was unaware of the present state of the art of Forth disassemblers. He put a lot of work into it; one learns by doing. At a short meeting on September 7, Wil Baden presented his text formatter, which was a model done in Pascal.

Northern California Chapter

At the October 22 morning FORML meeting, Doug Dillon presented code for a Modem-7 environment. Doug's code was written in Forth-83, using the Laxen and Perry implementation. Michael Stolowitz presented code on a B-tree and file-indexing method, also using F83. In the afternoon, Bob Reiling, Forth Convention Chairman, gave a review of the convention and reported an attendance of over 1200. Larry Forsley and Thea Martin, from Rochester, New York, spoke about the latest state of affairs at the Institute for Applied Forth Research, which they direct. Larry mentioned the next conference in Rochester, to be held June 5 through 9. The topic is to be Forth Applications, with one day devoted to real-time systems. Mike Perry gave a demonstration of a Forth-based CP/M BIOS generator, and discussed the virtues of Forth-83 in combination with the F83 implementation.

Support your local chapter!

John D. Hall is the Chapter Coordinator for the Forth Interest Group and is a consulting programmer.

Chapters in Formation

Here are more of the new chapters that are forming. If you live in any of these areas, contact one of these people and offer your support in forming a FIG chapter.

Contact:

Charles Samuels
7805 Linda Lane
Anchorage, AK 99502

Doug Dillon
California Cedar Products
P.O. Box 8449
Stockton, CA 95208
209/931-2448

Robert McFarland
DIGILOG Corp.
Box 3315
Ventura, CA 93006

Alan B. Cohen
14 Candlelight Dr.
Danbury, CT 06810

Alan H. Lake
Bankers First
P.O. Box 1332
Augusta, GA 30913

Michael Ham
3110 Alpine Ct.
Iowa City, IA 53340
319/337-1353

Manfred Peschke
Intersystems Mgmt. & Consult.
Story Hill Rd. RFD 3
Dunbarton, NH 03045

Gary Bergstrom
191 Miles Rd.
Chagria Falls, OH 44022

Steve Buffone
621 Center Ave.
Cuyahoga Falls, OH

Joel A. Neely
Interface Solutions, Inc.
Box 11167
Memphis, TN 38111

Jim Watson
801 Orleans
Corpus Christi, TX 78418

Pete Koza
9671 NE 122 Place
Kirkland, WA 98033

Zafar Essak
P.O. Box 46263
Vancouver, BC V6R 4G6
Canada

Norbert Heindl
reflecta-electronic gmbh
Berlichingstrasse 9
8540 Schwabach
West Germany

P.J. Reynolds
Murray & Roberts Bldg.
P.O. Box 4853
Cape Town 8000
South Africa

Fig Chapters

U.S.

• ARIZONA

Phoenix Chapter
Call Dennis L. Wilson
602/956-7678

• CALIFORNIA

Los Angeles Chapter
Monthly, 4th Sat., 11 a.m.
Allstate Savings
8800 So. Sepulveda Boulevard
Los Angeles
Call Phillip 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, 4th Wed., 7 p.m.
Fullerton Savings
Talbert & Brookhurst
Fountain Valley
Monthly, 1st Wed., 7 p.m.
Mercury Savings
Beach Blvd. & Eddington
Huntington Beach
Call Noshir Jesung
714/842-3032

San Diego Chapter
Weekly, Thurs., 12 noon.
Call Guy Kelly
619/268-3100 ext. 4784

• COLORADO

Denver Chapter
Monthly, 1st Mon., 7 p.m.
Call Steven Sarns
303/477-5955

• ILLINOIS

Fox Valley Chapter
Call Samuel J. Cook
312/879-3242

Rockwell Chicago Chapter
Call Gerard Kusiolek
312/885-8092

• KANSAS

Wichita Chapter (FIGPAC)
Monthly, 3rd Wed., 7 p.m.
Wilber E. Walker Co.
532 S. Market
Wichita, KS
Call Arne Flones
316/267-8852

• MASSACHUSETTS

Boston Chapter
Monthly, 1st Wed., 5 p.m.
Mitre Corp. Cafeteria
Bedford, MA
Call Bob Demrow
617/688-5661 after 7 p.m.

• MINNESOTA

MNFIG Chapter
Monthly, 1st Mon.
1156 Lincoln Avenue
St. Paul, MN
Call Fred Olson
612/588-9532

• MISSOURI

Kansas City Chapter
Call Terry Rayburn
816/363-1024

St. Louis Chapter
Monthly, 3rd Tue., 7 p.m.
Thornhill Branch of
St. Louis County Library
Call David Doudna
314/867-4482

• NEVADA

Southern Nevada Chapter
Suite 900
101 Convention Center Drive
Las Vegas, NV
Call Gerald Hasty
702/452-3368

• NEW JERSEY

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

• NEW YORK

New York Chapter
Monthly, 2nd Wed., 8 p.m.
Queens College
Call Tom Jung
212/432-1414 ext. 157 days
212/261-3213 eves.

Rochester Chapter
Monthly, 4th Sat., 2 p.m.
Hutchison Hall
Univ. of Rochester
Call Thea Martin
716/235-0168

Syracuse Chapter
Call C. Richard Corner
315/456-7436

• OHIO

Athens Chapter
Call Isreal Urieli
614/594-3731

Dayton Chapter
Twice monthly, 2nd Tues &
4th Wed., 6:30 p.m.
CFC, 11 W. Monument Ave.
Suite 612
Dayton, OH
Call Gary M. Granger
513/849-1483

• OKLAHOMA

Tulsa Chapter
Monthly, 3rd Tues., 7:30 p.m.
The Computer Store
4343 South Peoria
Tulsa, OK
Call Art Gorski
918/743-0113

• OREGON

Greater Oregon Chapter
Monthly, 2nd Sat., 1 p.m.
Computer & Things
3460 SW 185th, Aloha
Call Timothy Huang
503/289-9135

• PENNSYLVANIA

Philadelphia Chapter
Monthly, 3rd Sat.
LaSalle College, Science Bldg.
Call Lee Hustead
215/539-7989

• **TEXAS**

Dallas/Ft. Worth Metroplex Chapter
Monthly, 4th Thurs., 7 p.m.
Software Automation, Inc.
14333 Porton, Dallas
Call Marvin Elder
214/392-2802 or
Bill Drissel
214/264-9680

Houston Chapter
Call Dr. Joseph Baldwin
713/749-2120

• **VERMONT**

Vermont Fig Chapter
Monthly, 4th Thurs., 7:30 p.m.
The Isley Library, 3rd fl.
3rd Floor Meeting Room
Middleburynes, VT
Call Hal Clark
802/877-2911 days
802/452-4442 eves

• **VIRGINIA**

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

FOREIGN

• **AUSTRALIA**

Australia Fig Chapter
Contact: Ritchie Laird
25 Gibsons Road
Sale, Victoria 3850
051/44-3445

FIG Australia Chapter
Contact: Lance Collins
65 Martin Road
Glen Iris, Victoria 3146
03/29-2600

Sydney Chapter
Monthly, 2nd Fri., 7 p.m.
John Goodsell Bldg.,
Rm LG19
Univ. of New South Wales
Sydney
Contact: Peter Tregeagle
10 Binda Rd., Yowie Bay
02/524-7490

• **BELGIUM**

Belgium Chapter
Contact: Luk Van Loock
Lariksdreff 20
2120 Schoten
03/658-6343

• **CANADA**

Nova Scotia Chapter
Contact: Howard Harawitz
P.O. Box 688
Wolfville, Nova Scotia
B0P 1X0
902/542-7812

Southern Ontario Chapter
Monthly, 1st Sat., 2 p.m.
General Sciences Bldg,
Rm 312
McMaster University
Contact: Dr. N. Solntseff
Unit for Computer Science
McMaster University
Hamilton, Ontario L8S 4K1
416/525-9140 ext. 2065

• **COLOMBIA**

Colombia Chapter
Contact: Luis Javier Parra B.
Aptdo. Aereo 100394
Bogota
214-0345

• **ENGLAND**

Forth Interest Group -- U.K.
Monthly, 1st Thurs.,
7 p.m., Rm. 408
Polytechnic of South Bank
Borough Rd., London
Contact: Keith Goldie-Morrison
15 St. Albans Mansion
Kensington Court Place
London W8 5QH

• **ITALY**

FIG Italia
Contact: Marco Tausel
Via Gerolamo Forni 48
20161 Milano
02/645-8688

• **SWITZERLAND**

Contact: Max Hugelshofer
ERNI & Co. Elektro-Industrie
Stationsstrasse
8306 Bruttisellen
01/833-3333

• **TAIWAN**

Taiwan Chapter
Contact: J.N. Tsou
Forth Information Technology
P.O. Box 53-200
Taipei
02/331-1316

SPECIAL GROUPS

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

Baton Rouge Atari Chapter
Call Chris Zielewski
504/292-1910

FIGGRAPH
Call Howard Pearlmutter
408/425-8700

Vendors *(Continued from page 35)*

Triangle Digital Services, Ltd.
100A Wood St., Walthamstow
London E17 3HX England
01-520-0442 Telex 262284

Application Packages Only
See System Vendor Chart
for others

Curry Associates
P.O. Box 11324
Palo Alto, CA 94306
415/322-1463

InnoSys
2150 Shattuck Ave.
Berkeley, CA 94704
415/843-8114

Consultation & Training Only
See System Vendor Chart
for others

Bartholomew, Alan
2210 Wilshire Blvd. #289
Santa Monica, CA 90403
213/394-0796

Boulton, Dave
581 Oakridge Dr.
Redwood City, CA 94062

Brodie, Leo
17714 Kingsbury St.
Granada Hills, CA 91344
213/368-3677

Eastgate Systems Inc.
P.O. Box 1307
Cambridge, MA 02238

Girton, George
1753 Franklin
Santa Monica, CA 90404
213/829-1074

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

Harris, Kim R.
Forthright Enterprises
P.O. Box 50911
Palo Alto, CA 94303
415/858-0933

Intersystems Management
Computer Consultancy
Story Hill Rd. RFD3
Dunbarton, NH 03045
603/774-7762

Laxen, Henry H.
1259 Cornell Ave.
Berkeley, CA 94706
415/525-8582

McIntosh, Norman
2908 California Ave., #3
San Francisco, CA 94115
415/563-1246

Metalogic Corp.
4325 Miraleste Dr.
Rancho Palos Verdes, CA 90274
213/519-7013

Peschke, Manfred
Intersystems Mgmt. & Consult.
Story Hill Rd. RFD 3
Dunbarton NH 03045
603/774-7762

Petri, Martin B.
Computer Consultants
16005 Sherman Way
Suite 104
Van Nuys, CA 91406
213/908-0160

Redding Co.
P.O. Box 498
Georgetown, CT 06829
203/938-9381

Schleisiek, Klaus
Eppendorfer Landstr. 16
D 2000 Hamburg 20
West Germany
(040)480 8154

Schrenk, Dr. Walter
Postfach 904
7500 Karlstruhe-41
West Germany

Software Engineering
6308 Troost Ave. #210
Kansas City, MO 64131
816/363-1024

Softweaver
P.O. Box 7200
Santa Cruz, CA 95061
408/425-8700

Timin, Mitchel
3050 Rue d'Orlean #307
San Diego, CA 92110
619/222-4185

Technology Management, Inc.
1520 S. Lyon St.
Santa Ana, CA 92705
714/835-9512

FORTH System Vendors

(by Category)

(Codes refer to alphabetical listing e.g., A1 signifies AB Computers, etc.)

Processors

1802	C1, C2, F3, F6, L3
6502 (AIM, KIM, SYM)	R1, R2, S1
6800	C2, F3, F5, K1, L3, M6, T1
6801	P4, T1
6809	C2, F3, L3, M6, S11, T1
68000	C2, C4, D1, E1, F3, K1, T1
68008	P4, T1
8080/85	A5, C1, C2, F4, I5, L1, L3, M3, M6, R1, T3
Z80/89	A3, A5, C2, F3, F4, I3, L1, M2, M3, M5, N1, T3
Z80000	I3
8086/88	C2, F2, F3, L1, L3, M6
9900	E2, L3

Operating Systems

CP/M	A3, A5, C2, F3, I3, L3, M1, M2, M6, T3
CP/M-86	C2, F3
CP/M-68K	T1
FLEX	T1
RSX-11	F3

Computers

Alpha Micro	P3, S3
Apple	A4, F3, F4, I2, I4, J1, L4, M2, M6, M8, 02, 03
Atari	M6, P2, Q1, V1
Compaq	F3, M5
Cromemco	A5, M2, M6
DEC PDP/LSI-11	C2, F3, L2, S3
Heath-89	M2, M6, M7
Hewlett-Packard 85	
Hewlett-Packard 9826/36	C4
IBM PC	A8, C2, F3, L1, M5, M6, Q2, S9, W2
IBM Other	L3, W1
Kaypro II/Xerox 820	M2
Micropolis	A2, M2, S2
North Star	I5, M2, P1, S7
Nova	C5
Ohio Scientific	A6, B1, C3, O1, S6, T2
Osborne	M2
Pet SWTPC	A1, A6, B1, C3, O1, S6, T2, T5
Poly Morphic Systems	A7
TRS-80 I, II, III, XVI	I5, M2, M5, M6, S4, S5, S10
TRS-80 Color	A3, A8, F5, M4, S11, T1
Vector Graphics	M2

Other Products/Services

Applications	F3, P4
Boards, Machine	F3, M3, P4, R2
Consultation	C2, C4, F3, N1, P4, T3, W1
Cross Compilers	C2, F3, I3, M6, N1, P4, T1
Products, Various	A5, C2, F3, I5, S8, W2
Training	C2, F3, I3, P4, W1

FORTH Vendors (Alphabetical)

The following vendors offer FORTH systems, applications, or consultation. FIG makes no judgment on any product, and takes no responsibility for the accuracy of this list. We encourage readers to keep us informed on availability of the products and services listed. Vendors may send additions and corrections to the Editor, and must include a copy of sales literature or advertising.

FORTH Systems

A

1. AB Computers
   252 Bethlehem Pike
   Colmar, PA 18915
   215/822-7727
2. Acropolis
   17453 Via Valencia
   San Lorenzo, CA 94580
   415/276-6050
4. Applied Analytics Inc.
   8910 Brookridge Dr., #300
   Upper Marlboro, MD 20870
5. Aristotelian Logicians
   2631 E. Pinchot Ave.
   Phoenix, AZ 85016
7. Abstract Systems, etc.
   RFD Lower Prospect Hill
   Chester, MA 01011
8. Armadillo Int'l Software
   P.O. Box 7661
   Austin, TX 78712
   512/459-7325

B

1. Blue Sky Products
   729 E. Willow
   Signal Hill, CA 90806
2. Business Computing Press
   2210 Wilshire Blvd.
   Suite 289
   Santa Monica, CA 90403
   213/394-0796

C

1. Capstone Computing, Inc.
   5640 Southwyck Blvd., #2E
   Toledo, OH 43614
   419/866-5503
2. Chrapkiewicz, Thomas
   16175 Stricker
   East Detroit, MI 48021
3. CMOSOFT
   P.O. Box 44037
   Sylmar, CA 91342
4. COMSOL, Ltd.
   Treway House
   Hanworth Lane
   Chertsey, Surrey
   England KT16 9LA
5. Consumer Computers
   8907 La Mesa Blvd.
   La Mesa, CA 92041
   714/698-8088
6. Creative Solutions, Inc.
   4801 Randolph Rd.
   Rockville, MD 20852
   301/984-0262
7. Curry Associates
   P.O. Box 60324
   Palo Alto, CA 94306

E

1. Elcomp Publishing, Inc.
   53 Redrock Lane
   Pomona, CA 91766
   714/623-8314
   Telex 29 81 91
2. Elcomp-Hofacker
   Tegernseerstr. 18
   D-8150 Holzkirchen
   West Germany
   08024/7331
   Telex 52 69 73
3. Emperical Research Group
   P.O. Box 1176
   Milton, WA 98354
   206/631-4855
4. Engineering Logic
   1252 13th Ave.
   Sacramento, CA 95822
5. Eco Technologies
   1100 Larkspur Landing
   Circle #275
   Larkspur, CA 94939
   415/461-6121

F

1. Fantasia Systems, Inc.
   1059 The Alameda
   Belmont, CA 94002
   415/593-5700

3. FORTH, Inc.
2309 Pacific Coast Highway
Hermosa Beach, CA 90254
213/372-8493

4. FORTHWare
639 Crossridge Terrace
Orinda, CA 94563

5. Frank Hogg Laboratory
130 Midtown Plaza
Syracuse, NY 13210
315/474-7856

6. FSS
P.O. Box 8403
Austin, TX 78712
512/477-2207

H

1. HAWG WILD Software
P.O. Box 7668
Little Rock, AR 72217

I

1. IDPC Company
P.O. Box 11594
Philadelphia, PA 19116
215/676-3235

2. IUS (Cap'n Software)
281 Arlington Ave.
Berkeley, CA 94704
415/525-9452

3. Inner Access
517K Marine View
Belmont, CA 94002
415/591-8295

4. Innovatia Laboratories
5275 Crown St.
West Linn, OR 97068

5. Insoft
10175 S.W. Barbur Blvd.
Suite #202B
Portland, OR 97219
503/244-4181

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

J

1. JPS Microsystems, Inc.
361 Steelcase Rd., W.
Markham, Ontario
Canada L3R 3V8
416/475-2383

K

1. Kukulies, Christoph
Ing. Buro Datentec
Heinrichsallee 35
Aachen, 5100
West Germany

L

1. Laboratory Microsystems
4147 Beethoven St.
Los Angeles, CA 90066
213/306-7412

2. Laboratory Software
Systems, Inc.
3634 Mandeville Canyon
Los Angeles, CA 90049
213/472-6995

3. Lynx
3301 Ocean Park, #301
Santa Monica, CA 90405
213/450-2466

4. Lyons, George
280 Henderson St.
Jersey City, NJ 07302
201/451-2905

M

1. M & B Design
820 Sweetbay Dr.
Sunnyvale, CA 94086

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

3. Microsystems, Inc.
2500 E. Foothill Blvd., #102
Pasadena, CA 91107
213/577-1477

4. Micro Works, The
P.O. Box 1110
Del Mar, CA 92014
714/942-2400

5. Miller Microcomputer
61 Lake Shore Rd.
Natick, MA 01760
617/653-6136

6. Mountain View Press
P.O. Box 4656
Mountain View, CA 94040
415/961-4103

7. MCA
8 Newfield Ln.
Newtown, CT 06470

8. Metacrafts Ltd.
Beech Trees, 144 Crewe Rd.
Shavington, Crewe
England CW1 5AJ

N

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

O

1. OSI Software & Hardware
3336 Avondale Court
Windsor, Ontario
Canada N9E 1X6
519/969-2500

2. Offete Enterprises
1306 S "B" St.
San Mateo, CA 94402

3. On-Going Ideas
RD #1, Box 810
Starksboro, VT 05487
802/453-4442

P

1. Perkel Software Systems
1636 N. Sherman
Springfield, MO 65803

2. Pink Noise Studios
P.O. Box 785
Crockett, CA 94525
415/787-1534

3. Professional Mgmt. Services
724 Arastradero Rd., #109
Palo Alto, CA 94306
408/252-2218

4. Peopleware Systems Inc.
5190 West 76th St.
Minneapolis, MN 55435
612/831-0827

Q

1. Quality Software
6660 Reseda Blvd., #105
Reseda, CA 91335

2. Quest Research, Inc.
P.O. Box 2553
Huntsville, AL 35804
800/558-8088

R

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

S

1. Satellite Software Systems
288 West Center
Orem, UT 84057
801/224-8554

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

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

4. Sierra Computer Co.
617 Mark NE
Albuquerque, NM 87123

5. Sirius Systems
7528 Oak Ridge Highway
Knoxville, TN 37921
615/693-6583

6. Software Federation
44 University Drive
Arlington Hts., IL 60004
312/259-1355

7. Software Works, The
1032 Elwell Ct., #210
Palo Alto, CA 94303
415/960-1800

8. Spectrum Data Systems
5667 Phelps Luck Dr.
Columbia, MD 21045
301/992-5635

9. Stearns, Hoyt Electronics
4131 E. Cannon Dr.
Phoenix, AZ 85028
602/996-1717

10. Stynetic Systems, Inc.
Flowerfield, Bldg. 1
St. James. NY 11780
516/862-7670

11. Supersoft Associates
P.O. Box 1628
Champaign, IL 61820
217/359-2112

12. Sylmar Software
P.O. Box 44037
Sylmar, CA 91342

T

1. Talbot Microsystems
1927 Curtis Ave.
Redondo Beach, CA 90278
213/376-9941

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

3. Timin Engineering Co.
C/o Martian Technologies
8348 Center Dr. Suite F
La Mesa, CA 92041
619/464-2924

4. Transportable Software
P.O. Box 1049
Hightstown, NJ 08520
609/448-4175

V

1. Valpar International
3801 E. 34th St.
Tucson, AZ 85713
800/528-7070

W

1. Ward Systems Group
8013 Meadowview Dr.
Frederick, MD 21701

2. Worldwide Software
2555 Buena Vista Ave.
Berkeley, CA 94708
415/644-2850

3. Wycove Systems, Ltd.
P.O. Box 499
Dartmouth, NS B2Y 3Y8
Canada
902/469-9897

Z

1. Zimmer, Tom
292 Falcato Dr.
Milpitas, CA 95035

2. Ziggurat Software
P.O. Box 100
N. Salem, NH 03073

Boards & Machines Only
See System Vendor Chart
for others

Controlex Corp.
16005 Sherman Way
Van Nuys, CA 91406
213/780-8877

Datricon
7911 NE 33rd Dr., #200
Portland, OR 97211
503/284-8277

Golden River Corp.
7315 Reddfield Ct.
Falls Church, CA 22043

(Continued on page 33)

FORTH INTEREST GROUP

MAIL ORDER

	USA	FOREIGN AIR
☐ Membership in FORTH Interest Group and Volume V of FORTH DIMENSIONS	$15	$27
☐ Back Volumes of FORTH DIMENSIONS. **Price per each.** ☐ I ☐ II ☐ III ☐ IV	$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 Listings of fig-FORTH for specific CPU's and machines. The above manual is required for installation. Check appropriate box(es). **Price per each.**	$15	$18

☐ 1802 ☐ 6502 ☐ 6800 ☐ 6809 ☐ VAX ☐ Z80
☐ 8080 ☐ 8086/8088 ☐ 9900 ☐ APPLE II ☐ ECLIPSE
☐ PACE ☐ NOVA ☐ PDP-11 ☐ 68000 ☐ ALPHA MICRO

	USA	FOREIGN AIR
☐ "Starting FORTH, by Brodie. BEST book on FORTH. (Paperback)	$18	$22
☐ "Starting FORTH" by Brodie. (Hard Cover)	$23	$28

☐ PROCEEDINGS: FORML (FORTH Modification Conference)
- ☐ 1980, $25USA/$35Foreign
- ☐ 1981, Two Vol., $40USA/$55Foreign
- ☐ 1982, $25USA/$35Foreign

ROCHESTER FORTH Conference
- ☐ 1981, $25USA/$35Foreign
- ☐ 1982, $25USA/$35Foreign
- ☐ 1983, $25USA/$35Foreign Total $________

☐ STANDARD: ☐ FORTH-79, ☐ FORTH-83. $15USA/$18Foreign EACH. Total $________

☐ Kitt Peak Primer, by Stevens. An in-depth self-study book. $25 $35

☐ MAGAZINES ABOUT FORTH: ☐ BYTE Reprints 8/80-4/81
☐ Dr Dobb's Jrnl, ☐ 9/81, ☐ 9/82, ☐ 9/83
☐ Poplar Computing, 9/83 $3.50USA/$5Foreign EACH. Total $________

	USA	FOREIGN AIR
☐ FIG T-shirts: ☐ Small ☐ Medium ☐ Large ☐ X-Large	$10	$12
☐ Poster, BYTE Cover 8/80, 16"x22"	$ 3	$ 5
☐ FORTH Programmer's Reference Card. If ordered separately, send a stamped, self addressed envelope.	Free	

TOTAL $________

NAME________________ MS/APT________

ORGANIZATION________________ PHONE(   )________

ADDRESS________________

CITY________ STATE____ ZIP______ COUNTRY______

VISA#____________ MASTERCARD#____________

AMERICAN EXPRESS#____________ Card Expiration Date________
(Minimum of $15.00 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. 10/83

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

ROBERT [illegible]
2300 ST. FRANCIS DR
PALO ALTO, CA 94303

Address Correction Requested