Dimensions

Code Modules & Data Structures

Quick DP in Forth

WALK' on Bugs

Universal Stack Word

Fast Polynomial Evaluation
Mach1 is 32-bit FORTH-83 for the Mac.

Mach1 is a complete development system for writing multi-tasking applications, but it's so easy to use that it's being used to teach FORTH and 68000 assembly language at the university that spawned the Silicon Valley.

For developers, Mach1 is a multi-tasking programming environment with 'call' support for every Toolbox trap. It's subroutine-threaded for twice the speed of other Forths:

<table>
<thead>
<tr>
<th>Sieve</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>2-3 secs</td>
</tr>
<tr>
<td>compiled C</td>
<td>6-7 secs</td>
</tr>
<tr>
<td>Mach1</td>
<td>13 secs</td>
</tr>
<tr>
<td>other Forths</td>
<td>23 secs</td>
</tr>
<tr>
<td>Pascal</td>
<td>1270 secs</td>
</tr>
</tbody>
</table>

With the true assembler that's included, developers can use their unchanged MDS code in Forth. You can save progress on a project with the word 'workspace'. The new icon on the desktop will boot with all of your code as you left it. At the end of the project, the word 'turnkey' will create a stand-alone application (with only 16k of overhead for the multi-tasking operating system). Any application may be sold without licensing fees.

**Other features:** MacinTalk for words that speak, AppleTalk examples, stack notation and summary for every trap, headerless code, macro substitution, vectored I/O and ABORT, unlimited multi-tasking, named parameters, 400pg manual

Mach1 is only $49.95 w/ Switcher and Edit

Order from:

Palo Alto Shipping
PO Box 7430
Menlo Park, CA 94026

add $4 for S/H (CA Res add 6.5% sales)
call 800/44-FORTH to place VISA/MC orders
FEATURES

7 Making Numbers Pretty by Michael Ham
One difficulty many Forth novices have with number-formatting words is that they need a double-precision number as input. As part of reviewing the number-formatting words, the author also explains single- and double-precision, signed and unsigned numbers.

12 An Application of the Recursive Sort by Dr. Richard H. Turpin
The author of this note presents an application of his earlier sort code with a slight twist, using the Forth database model designed by Glen Haydon. The recursive stack-sort code allows one to print records in order of selected fields.

14 Quick DP in Forth by Len Zettel
It all started with a community church: the pledge system came under the care of a Forth programmer who was inspired to write a simple, line-oriented data processing program. He describes the process of creating this tool in terms most Forth beginners will understand.

16 WALK' on Bugs by David Franske
This powerful debugging tool allows one to see what happens on the Forth stack and what the interpreter executes, to test Forth programs bottom-up and to find bugs by halting applications in midstream to walk through suspect definitions with real data while using Forth’s full power to examine and change that data.

23 Code Modules and Data Structures by John S. James
How does one make data structures based on pointers, allowing a program itself to be ROMable, pure code? Two simple but useful implementations illustrate the use of CREATE...DOES> and ;CODE to define new classes of words in Forth.

25 A Universal Stack Word by Doneil Hoekman
Sometimes it’s difficult to avoid multiple, complex parameter stack operations — such as when dealing with mixed-mode arithmetic, floating-point, quad-precision or graphics routines. Welcome the novel word STACK, a new and promising solution to a dreary problem.

27 Fast Evaluation of Polynomials by Nathaniel Grossman
A Sixth FORML Conference speaker presented a utility that parses a stack diagram and an algebraic formula to generate stack operations that realize the formula. But the shortest possible sequence of stack permutations can be found without requiring the usual mental acrobatics.

35 FORML at Asilomar by Marlin Ouverson
At the annual conference of the Forth Modification Laboratory (FORML), Forth adepts contributed their expertise to the schedule of talks, poster sessions and working groups. Complete papers will soon be available in the published proceedings, but the following material sketches some of the highlights.

DEPARTMENTS

5 Letters

6 Editorial: “Elegant, Clever or Tricky?”

37 Advertisers Index

38 FIG Chapters
Don't Ask How We Did It!

No Visible Support Software

presents

an exclusive offer for Forth Dimensions Readers.

10 MB Paradise Hard Disk for the Apple Macintosh:

$699.-

20 MB Paradise Hard Disk for the Apple Macintosh:

$899.-

This offer expires March 31, 1986 so ACT NOW!

Send Check, Money Order, or VISA/Master Card number (CA Residents please include sales tax) to:

No Visible Support Software
PO Box 1344
2000 Center Street
Berkeley, CA 94704

Fine Print:
The Paradise Mac Disk comes with a 1 year manufacturer’s warranty. It includes a print spooler, a volume manager, a backup utility, a "Disk Doctor", and a nifty Show & Tell program.
Stack Your Locals

Editor:

"Local Definitions" (Forth Dimensions VI/6) seems like a useful utility. However, the system the code was written for is quite different from my Forth 79–83 hodgepodge.

My version uses an extra stack just below the disk buffers, which doesn’t waste any dictionary space. The only restrictions are that the number of buffers cannot be increased and the last 1K of memory is dedicated once the screens are loaded.

On my system, FIRST leaves the header address of the lower buffer in memory. N>LINK is defined if needed. I doubt if anyone would need 511 local words, so the stack can be reduced to something more appropriate. The stack words were adapted from Leo Brodie’s Thinking Forth.

Tom Shaw
Philadelphia, Pennsylvania

SCR # 20
// 20 LOCAL STACK 12/29/85
FIRST 1024 - CONSTANT LOCALS ( UNDER )
LOCALS 1024 ERASE ( BUFFERS)
LOCALS 1024 + CONSTANT LOCALS

: INIT.LOCALS
LOCALS LOCALS ! ; INIT.LOCALS

: ?ERR
IF . STACK ERR ? INIT.LOCALS
ABORT THEN ;

: PUSH.LOC ( N -- )
2 LOCALS +! LOCALS @ DUP LOCALS =
?ERR ! ;

: POP.LOC ( -- N )
LOCALS @ @ -2 LOCALS +! LOCALS @ LOCALS < ?ERR ;

SCR # 21
// 21 LOCAL WORDS 12/29/85

VARIABLE LINK.FROM

: LOCAL.START ( -- ) // 0 TO EXIT
INIT.LOCALS 0 PUSH.LOC ;

: LOCAL ( -- ) // LISTS FOR DELINKING
LATEST PUSH.LOC ;

: LOCAL.END ( -- )
LATEST N>LINK LINK.FROM ! POP.LOC BEGIN
LINK.FROM @ @ 2DUP = // PREV.NFA
IF SWAP N>LINK @ // RELINK
LINK.FROM @ ! DROP
ELSE
N>LINK LINK.FROM !
PUSH.LOC // NOT LOCAL RETURN IT
THEN POP.LOC DUP 0= // CHECK EXIT
UNTIL DROP ;

;S
LINK.FROM HAS LFA LAST GLOBAL WORD
: N>LINK PFA LFA ;

SCR # 22
// 22 TEST LOCAL WORDS 12/29/85

LOCAL.START

: 1WORD ." YOUR " ; LOCAL
: COW ;
: 2WORD ." NAME " ; LOCAL
: BULL ;
: 3WORD ." HERE " ; LOCAL
: NAME CR 1WORD 2WORD 3WORD ;

LOCAL.END

;S

Forth Can Flex

Dear Mr. Overson:

About a year ago, I wrote an implementation of Forth–83 for 6809 systems with Flex9. I recently converted the code for 6800 use, and I am releasing both versions for free distribution.

The implementation is quite complete, including a trace/debug function and a multi-tasker of the I/O-driven type, as well as the usual editor, assembler, etc. Anyone who would like a copy of the system, along with complete source code (in Forth) should send me two 5-inch diskettes with a reusable mailer and return postage. Please be sure to indicate which CPU version is wanted. Readers with access to CompuServe can also find the 6809 object code and limited documentation in the CLM SIG’s DL7.

Yours truly,
Wilson M. Federici
1208 N.W. Grant Corvallis, Oregon 97330

16-Bit Standard?

Dear Marlin,

Being deep into the process of converting my Z80 CP/M Forth to the Amiga 68000, I found Michael Hore’s letter (Forth Dimensions VII/3) on extending the Forth–83 Standard to machines with more than sixteen-bit addresses timely. I pretty much agree with his suggestions, but would like to offer a few remarks.

I agree that the stack cell size should be implementation dependent. But as I interpret the standard, that is already consistent with it. Although the standard specifies that addresses shall have a range from zero to 64K in the “standard address space,” there is no restriction on the size of the field in which addresses are placed, except that it must be at least sixteen bits. Presumably, that is the reason that standard programs are not allowed to address directly into the stacks.

Furthermore, for compilation addresses, there is not even a restriction to values less than 64K. In the “Definition of Terms” section, the standard specifies only that a compilation address shall be a “numerical value . . . which identifies” a definition. There is certainly no restriction on the size of the compilation address field in the dictionary — indeed, there is no detailed specification of the dictionary structure.

(Letters continued on page 6)
Elegant, Clever or Tricky?

Forth is malleable as a ball of wax. Its expressiveness allows finished code to reflect the creative thought processes — and perhaps even emotional characteristics — of the programmer. This is a more fundamental issue than whether one uses fixed- or floating-point arithmetic, which is a topic more easily understood and argued than the subtlety and elegance with which a truly “Forth-like” program operates. Why is it that some Forth programs look less intelligible than assembly language, while others are easily comprehended? How can one piece of Forth code be reminiscent of Los Angeles freeways at rush hour while another is as palatable as a picnic in the countryside?

To delve into lengthy comparisons of Forth to other languages is to get sidetracked. To wax philosophical is usually ineffectual and diffuse. At its best, good Forth programming seems to arise after its basics are mastered and one no longer mentally translates concepts from Pascal or C or any other language, and in which the solution is perceived first in the mind almost as a hologram that, when set down in Forth terms, satisfies all the standard criteria but is characterized by a natural directness that leads the reader to silently think, “Of course, that’s it!”

Defining what makes a Forth-like program is as difficult as generalizing about what makes a solution elegant instead of just clever or, worse, tricky. Writing a good example of this — except for those Zen programmers among us who are accustomed to quelling the mind and going straight to the heart of a problem to find the solution co-resident with it — may require discarding attempts that satisfy performance criteria but which do not go the one yard further into elegance. Fortunately, Forth supports iteration and modification in the same manner by which it supercedes, in a way, many of the differences between compiled and interpreted languages. If we let it, Forth allows us to arrive at a workable solution and then to see through to its logical underpinnings and express them in a simple, natural form.

Mimicry is the path chosen by so many companies in the computer industry that me-too’s and look-alikes have created fast fortunes, forced premature discounting and foundered otherwise well-run businesses. Leaders, however, especially those with evolutionary tools like Forth, look for significantly better solutions to old problems. They will also be looking at what Forth alone can do well.

In meeting the requirements of corporations that employ programmers, we must provide what professionals need and have come to expect in terms of tools, utilities, documentation and services. And if a vendor or consultant hopes to establish a permanent reputation, he must contribute to the stream of innovations that periodically infuse the marketplace with excitement, new direction and prosperity. This requires a two-way dialog with those we intend to serve, and the willingness to excel.

Those who go beyond just writing code that runs, who create a body of work that exemplifies Forth-like programming in every way, will be establishing the same mental climate required of all leaders and surely will translate that into material success. Let’s cultivate the qualities of leadership within our businesses as carefully as a sculptor molds his wax model before casting it in precious metals.

—Martin Ouverson
Editor

Letters continued from page 5

(Letters continued from page 5)
ture at all. In my 68000 implementation, the compilation address field has a variable size, with a lower bound in practice of four bytes which occurs, for example, in the special case when it contains a twenty-four-bit address in a thirty-two-bit field.

Although the above do not appear to be violations of the standard, some violations are inevitable in a more than 64K environment; and I agree with Michael Hore that Forth is going to be left behind if it does not accommodate.

I think the simplest solution to the problem of parameter and address size is to make it (judiciously) implementation dependent, with a lower bound of sixteen bits. In my case, for the 68000 I simply multiply by two most of the standard-parameter sizes in arithmetic and memory transfer operations. Thus, +, @ and 1 will operate on thirty-two-bit quantities, and D+, 2@ and 2! will work on sixty-four bit quantities. I keep the size for words like C@, C! and CMOVE at eight bits, and I introduce new words W@, W+, etc. for sixteen-bit sizes. I expect that most programs written in a sixteen-bit environment will still work.

My attitude towards the standard is that it should be taken very seriously as a common starting point to be violated only for good reasons, but then without hesitation or regret. That the future of personal computing belongs to larger address spaces has been clear to a lot of people for some time. The cheapness of memory has now confronted us with that reality, and for Forth it has become a case of grow or die. In my view, it offers the community a real opportunity to rethink and develop the structure of Forth; I hope we won’t be too timid about it. For example: up with long, dynamic strings! Down with screens and blocks! In any case, let’s rally to Michael Hore’s call for discussion.

Sincerely,
David N. Williams
Ann Arbor, Michigan
Making Numbers Pretty

Michael Ham
Santa Cruz, California

The number-formatting words <# # $S SIGN and #$ seemed strange when I first began to use Forth, and I approached them gingerly. In this article, I invite you to play with them to in a nice format: the number 12,345.67.

A high-order bit of 1 in a single-precision number signals a minus sign. A high-order bit of 1 in a double-precision number signals a minus sign by itself. The double-precision number's high-order bits (contributed by the zero we put on the stack) are all zero.

Try playing with the new .BITs for a while. (Things will be easier for you if you edit these examples into a disk file so that they can be easily retrieved and revised.) You can break up the string of zeroes and ones by putting the word SPACE into the definition of 16BITs where you want a space. For example, redefine 16BITs this way (and then reload the definitions):

```forth
: 16BITs ( d - )
   <##### SPACE ##### 3 SPACES ##### SPACE #####> TYPE ;
```

The spaces display the sixteen-bit string in bytes (eight bits) and nibbles (four bits). Now that we can see the bit pattern of a number, try looking at some positive and negative numbers. You will find –1 and 65535 have the same bit pattern, for example, as does –45 and 65491. You will also note that such “two-way” numbers consist only of numbers greater than 32767, for it is only at 32768 that the high bit is turned on. The high-order 1 bit is what leads to the two interpretations.

We have just seen how to create an unsigned extension from a single-precision number to a double (that is, how to treat the single-precision’s high order bit as part of the number and not as a sign): we simply put a zero on the stack above the number. In the result-
For Programming Professionals:  
an expanding family of compatible, high-performance,  
Forth-83 Standard compilers  
for microcomputers

For Development:  
Interactive Forth-83 Interpreter/Compilers  
• 16-bit and 32-bit implementations  
• Full screen editor and assembler  
• Uses standard operating system files  
• 400 page manual written in plain English  
• Options include software floating point, arithmetic  
coprocessor support, symbolic debugger, native code  
compilers, and graphics support

For Applications: Forth-83 Metacomplier  
• Unique table-driven multi-pass Forth compiler  
• Compiles compact ROMable or disk-based applications  
• Excellent error handling  
• Produces headerless code, compiles from intermediate  
states, and performs conditional compilation  
• Cross-compiles to 8080, Z-80, 68000, and 6502  
• No license fee or royalty for compiled applications

Support Services for registered users:  
• Technical Assistance Hotline  
• Periodic newsletters and low-cost updates  
• Bulletin Board System

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

0 CONSTANT US>D

We now will define the extension operator S>D for signed numbers. S>D  
will assume that the high-order bit is a sign. If that bit is on, then the  
number is assumed to be negative; if it’s off, the number is positive. For  
positive numbers, the extension to double-precision is exactly as before:  
put a zero above the number on the stack.

The easiest way to see whether the  
high bit is on is to use the operator 0<.  
0< leaves a false flag (zero, or all bits  
off) if the number is positive (high bit  
off) and a true flag (-1, or all bits on) if  
the number is negative (high bit on). To  
extend the number to double-precision,  
we replicate its high-order bit across  
the cell placed on the top of the stack:  
if the high bit is on, we want the cell on  
top of the stack to be all bits on; if the  
high bit is off, the cell on top of the  
stack should be all bits off. We thus  
arrive at this simple definition:

:S>D (n-d) DUP 0<;

This definition assumes an 83-Standard Forth. If your Forth is 79-Standard,  
the true flag is 1 instead of -1: one  
bit on instead of all bits on. You can  
correct the definition by ending it with  
-1 * or DUP IF DROP -1 THEN. (Which is  
faster on your system?)

We now can extend either signed or  
unsigned numbers to double-precision.  
And of course we may be dealing with  
double-precision numbers to begin with. In any case, we can now assume  
that we have a double-precision num-  
ber to print. Let’s assume that the  
number may have been scaled — that  
is, although it is stored as an integer,  
some digits are in fact to the right of
the decimal point. We'll define a variable to keep track of the number of decimal places in the number being printed:

```
VARIABLE DECIMALS
: PLACES ( n -- ) DECIMALS ! ;
```

2 PLACES thus indicates that the number has two decimal places: two digits to the right of the decimal point.

To print the commas, we need to know also the number of digits to the left of the decimal point. To get this number we might, for example, divide the number by ten until the quotient is zero. The number of divisions is the number of digits.

The 83-Standard arithmetic operators for double-precision numbers are a mixed bag: D+ and D- use double-precision operands and produce double-precision results, while UM* and UMMOD are mixed operations, with the one giving a double-precision result and the other a single-precision. If you want to to divide a double-precision number and get a double-precision result, you must work with the particular Forth you are using.

Many vendors do provide a larger set of operators; for example, you might find an operator like M/D for division. This operator takes a double-precision dividend and a single-precision divisor (and thus is a mixed operation), and it returns a double-precision result, as indicated by the D in its name. Using M/D we can develop this definition:

```
VARIABLE DIGITCOUNT
: #DIGITS ( d -- )
  DABS 1 DIGITCOUNT !
  BEGIN 10 M/D 2DUP OR
  WHILE 1 DIGITCOUNT +! REPEAT
  2DROP DIGITCOUNT @ ;
```

DIGITCOUNT holds the count as it is accumulated. You can avoid using a variable, but the cost in terms of stack swapping operations (and complexity of the definition) isn't worth it.

The definition first takes the absolute value of the double-precision number. The number of digits in the absolute value is the same as in the signed value, of course. If we didn't take the absolute value, we would run into a problem when the number was negative: a negative number divided by ten produces a negative quotient, which will never reach zero, regardless of the number of divisions (the largest it will get is -1). The 2DUP OR leaves all zero bits (binary 0) on the stack only if the double-precision number has no one bits — 2DUP OR is thus the double-precision equivalent of 0=. The loop will thus increment the count of digits until the quotient is zero. The initialization of DIGITCOUNT with a one takes care of the case in which the number is less than ten to begin with and the loop is therefore exited before DIGITCOUNT is incremented for the first time.

If your Forth has no operator that produces double-precision quotients, use some other approach. One way, somewhat heavy-handed but fast, is to compare the number to successive powers of 10 - 1 (9, 99, 999, 9999, etc.), and stop as soon as the number is less than ten. The number of comparisons is the number of digits. (Remember: the number is assumed to be double-precision.)

```
: 2, ( d -- ),
CREATE NINES
9. 2. 99. 2. 999. 2. 9999. 2. 99999. 2. 999999. 2.
9999999. 2. 99999999. 2,

: #DIGITS ( d -- )
  DABS 1 DIGITCOUNT !
  BEGIN 2DUP DIGITCOUNT @
  1 - 4 * NINES + 2D D>
  WHILE 1 DIGITCOUNT +! REPEAT
  2DROP DIGITCOUNT @ ;
```

With the number of digits and the number of decimal places, it is easy to calculate the number of commas. If we can assume that #DIGITS has been executed, we can use this definition:

```
: #COMMAS ( - # )
  DIGITCOUNT @ DECIMALS @
  - 3 MOD SWAP 0 + 0 MAX ;
```

The name, succinct as it is, might be criticized as cryptic. Change it to NUMBER_OF_COMMAS or COMMAS if you wish. The definition also does arithmetic using the 83-Standard true flag (-1). You can make the code...
also available for other popular
mini and micro computers.

IBM PC*

PolyFORTH II offers IBM PC users:
• Unlimited control tasks
• Multi-user capability
• 8087 mathematics co-
processor support
• Reduced application
development time
• High speed interrupt
handling

Now included at no extra cost:
Extensive interactive GRAPHICS
SOFTWARE PACKAGE! Reputed
to be the fastest graphic package
and the only one to run in a true
multi-tasking environment, it
offers point and line plotting,
graphics shape primitives and
interactive cursor control.

PolyFORTH II is fully supported
by FORTH, Inc.'s:
• Extensive on-line
documentation
• Complete set of manuals
• Programming courses
• The FORTH, Inc. hot line
• Expert contract programming
and consulting services

From FORTH, Inc., the inventors
of FORTH, serving professional
programmers for over a decade.
Also available for other popular
mini and micro computers.
For more information contact:

FORTH, Inc.
2309 Pacific Coast Hwy.
Hermosa Beach,
CA 90254
213/372-8493
RCA TELEX: 275182
Eastern Sales Office
1300 N. 17th St.
Arlington, VA 22209
703/525-7778

*IBM PC is a registered trademark of International
Business Machines Corp.

We must also cover the other possi-
bility that the number is signed and
thus may be negative. We can put SIGN
into the number wherever we want the
minus sign to appear — before, after,
or in the middle. (SIGN prints only
the negative sign; if the number is positive,
no plus is printed.) Let’s put the sign in
front of the number.

SIGN uses the sign of the number on
top of the stack. Since in a double-
precision number the sign is carried by
the top cell, we can save a copy of just
this cell. The number-print word re-
quires that the absolute value be used.
So we get a second definition:

: PRT# ( d – adr cnt )
DUP 2DUPS <# DECIMALS @?DUP
IF 0 DO # LOOP ASCII . HOLD THEN
#S ?DUP
IF 0 DO # # ASCII , HOLD THEN
#S # > ;

UPRT# first counts the number of
digits. It then begins converting the
number into a string that can be printed.
Since numbers are converted beginning
with the low-order (rightmost) digits,
we are first concerned with the
digits to the right of the decimal point.
DECIMALS holds the count of decimals.
If it is zero, then the count (undupli-
cated by ?DUP) serves as a false flag,
and the IF clause is skipped. If the
count is not zero, we loop as many
times as there are digits to the right of the
decimal, executing # in each iter-
ation of the loop, thus formatting the
correct number of digits. ASCII , HOLD
then puts the decimal point into the
string. (If your FORTH doesn’t include
the word ASCII you can use 46 HOLD,
fourty-six being the decimal value of
the period character.)

The same approach is used for the
commas. If no commas are required,
we simply print the (rest of) the num-
ber with #S. If commas are needed, we
loop once for each comma, outputting
three digits and one comma each time
through the loop. (You can substitute
for ASCII , the actual value forty-four.)

#S finishes the number (without
leading zeroes). Because #> puts the
address and count on the stack, UPRT#
leaves the stack ready for TYPE to print
the number.
cell is not negative, then you must
determine whether the double-precision number is zero or not. If it is zero,
put a zero on the return stack; otherwise,
put a positive number on the return stack. (I leave this activity as an
exercise for you.)

+SIGN uses two IF statements so that
no plus is printed when the number is
zero. The second IF uses the number from the return stack as the flag: zero
will act as a false flag and thus will not get the plus sign.

The final enhancement is to print the number flush-right in a field of speci-
fied width — so that the number-print-
ning word works as .R and U.R. In those,
the number immediately before the print word specifies the width of the field,
and we mimic that pattern in the following definitions:

: Ud.r ( ud # - )
   >R UPRT# R> OVER -
   SPACES TYPE ;
: d.r ( d # - )
   >R PRF# R> OVER -
   SPACES TYPE ;
: r ( n # - )
   SWAP S>D ROT d.r ;
: U.r ( u # - )
   SWAP US>D ROT d.r ;

The names follow the pattern of the number-printing words already pre-
sent, but the names are lower case both to distinguish them and to suggest that
these definitions will print the numbers with a more user-friendly appearance
(the use of lower case being less intimidating than shouting at the user IN
UPPER CASE).

In the double-precision print words, the width of the field (represented by #
in the stack comment) is saved on the return stack until the number is ready
to print. It is then retrieved, the charac-
ter count of the number to be printed is
subtracted from it, and that many spaces are printed before the number is
printed. For example, if we are printing a six-character number string (includ-
ing any sign, commas, or decimal points) in a ten-character field, the
definitions above print four spaces
before printing the number, thus making it flush-right.

The two single-precision words con-
vert the single-precision number to
double-precision, return the field width
to the top of the stack, and then call
the double-precision word. Since the choice of the conversion word deter-
mines the interpretation of the sign bit,
we can use d.r for both words.

Some examples of phrases using
these words:

\begin{verbatim}
TOTAL @ 12 .r
SAMPLES @ 12 U.r
CASH 2@ 25 d.r
VOLUMES 2@ 15 Ud.r
\end{verbatim}

In each example, a number is fetched
from a variable. If the number was
fetched from a VARIABLE and thus is
double-precision, either d.r or Ud.r
is used depending on whether numbers in
excess of 2,147,483,647 are to be inter-
preted as negative or positive. (If all of
the numbers are going to be less than
2,145,483,648, then it won't make any
difference which of the two you use.) If
the number is single-precision, r or U.r
is used depending on whether the number
is to be interpreted as signed or
unsigned.

\begin{quote}
\textit{Michael Ham is a freelance pro-
grammer, systems designer and writer
in Santa Cruz, California. This article
is from a book in progress. Copyright © 1986 by Michael Ham.}
\end{quote}
An Application of the Recursive Sort

Dr. Richard H. Turpin
Stockton, California

In the July/August 1983 issue of Forth Dimensions, I presented code which sorts data on the stack. In this note I present an application of the sort code with a slight twist.

Using the database model designed by Glen Haydon and presented in Forth Dimensions (III/2), I have written a gradebook system for use in my classes at the university. The data for each student are stored as a record in the GRADES file. Each record is composed of a number of fields, such as MT1 (mid-term number one), FINAL (final exam), etc.

For certain course reports, I want to print out student records in order of selected score fields. To accomplish this I make use of the recursive sort on the stack in the following manner. I first load onto the stack, in numeric order, the student record numbers (e.g., two through twenty-one for a class of twenty students; record numbers zero and one are used for reference scores and weights). I then call a modified version of SORT to rearrange the record numbers on the stack based on a comparison of the corresponding field data (the original version of SORT compared the original stack data). Then, to generate the report I need only read and print the student records in the order of the record numbers left on the stack.

Screen 39 gives the resultant code. LOAD-STACK places record numbers onto the stack in numeric order. BUBBLE is called by SORT a sufficient number of times to sort the record numbers on the stack by reading and comparing the data of each record as specified by FIELD (lines five and six). Finally, SORTED puts it all together to provide the desired function. For example, by typing FINAL SORTED the student records will be displayed sorted, based on the final exam scores.

The code in this application is written using F83. #RECORDS returns the
number of student records in the file. The word READ in lines five and thirteen reads and makes active a student record using the top number on the stack as the record number. SUMMARY outputs student data in a particular summary format. SET.FIELD takes a parameter from the stack and stores it in memory to designate the data field being referenced for a given application. FIELD returns the address of the selected data field for the active student. These words are part of the gradebook application. All other words are standard equipment of F83.

To illustrate this application of SORT, a dummy class roster was generated. (Note: this is not a class of dummies, although most of the names and all of the identification numbers are fictitious.) Column headings are not included. The first listing shows the normal, unsorted summary of scores. The second listing gives the data sorted on midterm number one (score column four). Finally, the last listing provides the same class data sorted on the percent total score (rightmost column). In each case, the command used to obtain the summary is shown above the output.

| number of student records in the file. The word READ in lines five and thirteen reads and makes active a student record using the top number on the stack as the record number. SUMMARY outputs student data in a particular summary format. SET.FIELD takes a parameter from the stack and stores it in memory to designate the data field being referenced for a given application. FIELD returns the address of the selected data field for the active student. These words are part of the gradebook application. All other words are standard equipment of F83.

To illustrate this application of SORT, a dummy class roster was generated. (Note: this is not a class of dummies, although most of the names and all of the identification numbers are fictitious.) Column headings are not included. The first listing shows the normal, unsorted summary of scores. The second listing gives the data sorted on midterm number one (score column four). Finally, the last listing provides the same class data sorted on the percent total score (rightmost column). In each case, the command used to obtain the summary is shown above the output. |
Quick DP in Forth

Len Zettel
Trenton, Michigan

I read somewhere that the two really practical applications for a home computer are to play games and to give your kids some idea of what a computer is all about. Certainly, the main reason I got mine was to have fun, and I have. In particular, learning my way around Forth has been an absorbing and enlightening experience. Still, I must confess that I am no longer pure: I have now used my computer at a task that was very useful to me, and Forth had a major part to play.

It all started when I was wrestling with the canvass records for our church. Reasonably enough, this was an all-manual system whose backbone was a set of about 220 pledge cards with the relevant information filled out on them. I would hand the cards to canvassers and keep a stub so I knew who had what. The canvassers would visit the families, return the cards to the church office and phone the results (or lack thereof) to me. I would then make notes on the stubs, sorting them into two bundles — finished and outstanding.

It didn’t take long for me to get tired of doing through the stubs and totting up results with a pocket calculator (with bad key bounce). Doggonit! I had a computer, didn’t I? (Okay, so it was a VIC 20 with 16K expansion — that should be plenty for this.) The only real constraint was that I wanted something useful that would require no more effort than this trivial but onerous chore.

To tackle this in the best Forth tradition, did I have anything sitting around that could be part of the solution? Well, I had HES FORTH adjusted to run with a disk system, which gave me a pretty good full-screen editor. If I could keep my records to sixty-four characters, each record could be a line on a Forth screen, and I could put anything I wanted on any line by simple keyboard entry. I wanted to get a printout, each line of which would look something like:

(139) 470 25700 Smith, Sarah and Harold

139 was the identifying number that keyed my stub with the pledge card. 470 would be the amount the Smiths pledged and 25700 would be the cumulative total pledged from the start of the file.

With proper use of the editor, it would be no problem to add entries as they came in and keep the whole thing in alphabetical order. Just doing a LIST wouldn’t get my calculations done, though, and manually editing the running total would be a worse pain than using the calculator. However, if I surrounded the character string with " (dot-quote) at the beginning and " (quote) at the end, then LOAD would get me the strings printed out at a very acceptable overhead of six characters per record.

Now all we have to figure out is how to do the arithmetic. How about a word +. (plus-dot, or add-print)? We could do something like this:

```forth
:+. (total n1 -- total)
   DUP . + DUP .

Then, a record would look like this:

"(139)" 470 +.
"Smith, Harold and Sarah" CR
```

We could fix it so there would be a zero on the stack before the first record was processed, to ensure the running total would start out correctly. Looks good, but there are a couple of problems.

First, a simple . for the print means that as the numbers get bigger the running total and the names will print farther to the right. Second, there will be problems when the total climbs over 32,767 (and it had better, or the church would be in real trouble). So maybe we should keep a double-precision total and use .R to print the results. Maybe some spaces would pretty them up, too. So, for a second try we have:

```forth
:+. (n1 n2 n3 -- n3 n2 n1)
   >R ROT 1 + DUP 4.R
   ROT ROT 2 SPACES
   R> DUP 5.R 2 SPACES
   S->D D+ 2DUP
   6 D.R 2 SPACES ;
```

That met the original specifications and worked quite well. I keyed my data into two groups of screens, one for cards completed and one for cards outstanding. Following Thinking Forth, I created a couple of screens to LOAD the data screens in the correct order. This made it easy to create new screens between existing ones as the file expanded. When another card came in, I could use the editor to move the line from the cards-outstanding screen to the cards-completed screen, editing the pledge amount if it had changed. Then, with a simple execution I had updated totals on my next printout. Glorious!

Then came a couple more refinements. I got tired of counting lines to see how many records were in the file. How could I get a record number to print — without much work? Very simply, and without touching the data records at all! We just have to get +., to manipulate another stack quantity, incrementing it by one and printing it each time it executes. This brings us to:

```forth
:+. (n1 d1 n2 -- n3 d2)
   >R ROT 1 + DUP 4.R
   ROT ROT 2 SPACES
   R> DUP 5.R 2 SPACES
   S->D D+ 2DUP
   6 D.R 2 SPACES ;
```

Things went swimmingly for a while, until like the fisherman’s wife I found I had another wish — er, need. Now I wanted to know how many entries on the file had zero amounts. Simple: we define VARIABLE ZERO-AMOUNTS and rework faithful old +., one more time:
And that is where we are today. Not a profound application, but it was fun to work out. It filled a real need and I think it neatly illustrates many of the virtues of Forth. Note the complete absence of a big, clumsy conventional program reading in a data file and trying to decide what to do with it. We just prepare (easily!) the right kind of file, and then let the text interpreter itself process.

Then there is the natural factoring of the problem. Three different versions of +, and we didn’t have to touch the data screens at all. Either Forth is the neatest way yet devised to get useful results out of a von Neumann machine, or I have become a hopeless crank. May the Forth stay with us all!

**HS/FORTH**

- Full 8087 high level support.
- Full range transcendentals (\(\tan, \sin, \cos, \arctan, \log, \exp, \text{exponentials}\)).
- Data type conversion and I/O parse/format to 18 digits plus exponent.
- Complete Assembler for 8088, 80186, and 8087.
- String functions - (LEFT RIGHT MID LOC COMP XCHG JOIN)
- Graphics & Music
- Includes Forth-79 and Forth-83
- Full megabyte - programs or data
- Fully Optimized & Tested for: IBM-PC XT AT and JR COMPAQ and TANDY 1000 & 2000 (Runs on all true MSDOS compatibles!)
- Compare BYTE Sieve Benchmark Jan 83 HS/FORTH 47 sec BASIC 2000 sec with AUTO-OPT 9 sec Assembler 5 sec other Forths (mostly 64k) 55-140 sec
- FASTEST FORTH SYSTEM AVAILABLE.
- TWICE AS FAST AS OTHER
- MEGABYTE FORTHS! (TEN TIMES FASTER WHEN USING AUTO-OPT)
- HS/FORTH, complete system only: $270.

**PRIME FEATURES**

- Execute DOS level commands in HS/FORTH, or execute DOS and BIOS functions directly.
- Execute other programs under HS/FORTH supervision. (editors debuggers file managers etc)
- Use our editor or your own.
- Save environment any time as .COM or .EXE file.
- Eliminate headers, reclaim space without recompiling.
- Trace and decompile.
- Deferred definition, execution vectors, case, interrupt handlers.

**MCA**

**FIG-Forth for the Compaq, IBM-PC, and compatibles. $35**

Operates under DOS 2.0 or later, uses standard DOS files.

Full-screen editor uses 16 x 64 format. Editor Help screen can be called up using a single keystroke.

Source included for the editor and other utilities.

Save capability allows storing Forth with all currently defined words onto disk as a .COM file.

Definitions are provided to allow beginners to use *Starting Forth* as an introductory text.

Source code is available as an option

**A Metacomplier on a host PC, produces a PROM for a target 6303/6803**

Includes source for 6303 FIG–Forth. Application code can be Metacompiled with Forth to produce a target application PROM. **$280**

**FIG–Forth in a 2764 PROM for the 6303 as produced by the above Metacomplier. Includes a 6 screen RAM–Disk for stand–alone operation. $45**

An all CMOS processor board utilizing the 6303. Size: 3.93 x 6.75 inches. Uses 11–25 volts at 12ma, plus current required for options. **$240 – $360**

Up to 24kb memory: 2kb to 18kb RAM, 8k PROM contains Forth. Battery backup of RAM with off board battery.

Serial port and up to 40 pins of parallel I/O.

Processor bus available at optional header to allow expanded capability via user provided interface board.

**Micro Computer Applications Ltd**

8 Newfield Lane
Newtown, CT 06470
203-426-6164

Foreign orders add $5 shipping and handling. Connecticut residents add sales tax.
WALK’ on Bugs

David Franske
Vancouver, British Columbia Canada

This is a powerful Forth debugging tool which enables you (1) to see what happens on the Forth stack and what the interpreter executes, by walking through word definitions (lots of help to novices), (2) to test Forth programs bottom-up by verifying each word definition, and (3) to find bugs in large application programs that are doing strange things, by halting them in mid-stream to walk through suspect definitions with real data while using Forth’s full power to examine and/or change that data.

The new word WALK’ encapsulates all these functions. The apostrophe, analogous to the Forth word ‘ (tick) signifies that the next item in the input stream is the parameter. WALK’ may be executed directly or edited into a definition just before a suspect word.

If you have not yet had a look at my sample WALK’ of FLUSH, take a look. If you try to walk a primitive, you get the stack dumped out both before and after the primitive executes; but when a secondary is walked, the system goes into break mode at the first word in the secondary’s definition. When the system breaks, (1) the current value of the interpretive pointer (IP) is printed; then, (2) the depth of the stack is printed, (3) the contents of the stack and the word about to execute are printed using a format similar to the way you would type them in (i.e., with the top item on the stack farthest to the right), and (4) the system waits for your response.

As a response, you may enter any normal Forth commands or a blank line. If you enter Forth commands, they will be executed and “BOK” will be printed to remind you that the system is in this break mode. If you enter a blank line, the system will execute the presently pending instruction and break before the next instruction. If the pending instruction is a secondary, the program will not nest down a level unless you use the special, added command DOWN. This keeps you from being inundated with a lot of useless information. If you wish to resume the definition being stepped through at full speed, type GO and press the return key twice. When you want to terminate the application, just type ABORT.

The initial ideas for this program came from Leo Brodie’s article “Add a Break Point Tool” (Forth Dimensions, V/1) and from my experience with fig-FORTH’s 6502 implementation, which has a trace feature that does not give a readable printout and which nests down into every secondary it finds. You must have a good understanding of threaded interpreters to comprehend how this program operates. WALK’, when invoked, changes a couple of the final jumps in the regular Forth interpreter to point to its extra routines. At their ends, each of the added routines jumps to where the routine they are tacked onto would have gone if there wasn’t the detour. When-to-BREAK decisions center around the value in NEST#. The three routines MORECOLON, MOREDOES and MORENEXT manipulate the value of NEST#. When the value in NEST# is zero, the MORENEXT machine language routine calls the high-level Forth routine BREAK. The routine MOREEXIT removes the added routine with VECTORS-OUT when the return stack has become higher than the value stored in SPSAVE. This is why just doing an ABORT cleans up the whole thing.
The **ForthCard**

**STAND ALONE OPERATION**

**STD BUS INTERFACE**

**EPROM/EEPROM PROGRAMMER**

**RS-232 I/O**

**PARALLEL I/O**

**ROCKWELL FORTH CHIP**

The ForthCard provides OEMs and end users with the ability to develop Forth and assembly language programs on a single **STD bus compatible card**.

Just add a CRT terminal (or a computer with RS-232 port), connect 5 volts and you have a **self contained Forth computer**. The STD bus interface makes it easy to expand.

Download Forth source code using the serial port on your PC. Use the **onboard EPROM/EEPROM programming** capability to save debugged Forth and assembly language programs. Standard UV erasable EPROMs may also be programmed with an external Vpp supply.

**Evaluation Unit** $299

Part #STD65F11-05 includes: ForthCard, Development ROM, 8Kbyte RAM, Manuals

**OEM Version as low as** $199

Part #STD65F11-00 does not include memory or manuals

**NEW! Options and Application Notes**

- Electrically Eraseable PROMs (EEPROMs)
- FREEZE the dictionary in EEPROM (save in non-volatile memory, to be restored on power up)
- Download Software for your IBM PC or CP/M
- Non-Volatile CMOS RAM with battery 2K, 8K, optional Clock/calendar
- Fast 2MHz clock (4MHz crystal)
- Disk Controller Card (5¼”)
- Self Test Diagnostics
- Parallel printer interface

**Ask about our ForthBox™**

A complete STD bus oriented system including the ForthCard, Disk Controller, Disk Drive(s), STD Card Cage, Cabinet and power supply.

**CALL TODAY FOR COMPLETE INFORMATION!**

**HiTech Equipment Corporation**

9560 Black Mountain Road
San Diego, CA 92126
(619) 566-1892
This program has a few weaknesses: the self-modifying code is very dangerous and a bad practice, and it is not completely transparent to other programs. Generally, the program will mess up with I/O and routines that use data at HERE and PAD. A lot can be done to make this program more transparent if you want to go to the trouble of saving and restoring data areas. One other nice improvement, if you have a bigger screen, is to modify the procedure .STACK to print more stack values.

Another good extension to the program would be to integrate it with a Forth decompiler. This requires only the redefinition of .WORD.

Sorry, but this program will only work on systems where the Forth kernel is resident in RAM. If you are going to implement this, be careful to find the correct jump vectors. To find the right vector location for NEXTJ on 6502 systems, type in:

```
NEXT 23 + w 1- = .<CR>
```

If your version of Forth corresponds to the fig-FORTH model, you will get a true flag. Otherwise, if you get a false flag, adjust the offset 23 until you get a true flag. In non-6502 systems, it may be necessary to modify the actual opcode of the last instruction in the NEXT routines to perform a jump to the MORENEXT instead of an indirect jump on W. The other vectors may be found in a similar way: the EXITJ and COLONJ vectors will each equal NEXT and the DOESJ will equal PUSH.

```
create moreexit
nest# inl,
next jmp,
( extension to doesj )

create moreexit
nest# inl,
push jmp,
( extension to is and exit )

create morenext ( addr -- )
nest# lda, or
if
 vectos-out jsr,
 ip lda, 1save sta,
 ip+ 1+ lda, 1save sta,
 w 1+ 1+ lda, 1save sta,
 x 1+ 1+ lda, 1save sta,
 else
 status
 "primitive"
then

create dup
 o o lda, i + i sta,
next jmp

create hi
 i + i lda, + sta,
 + lda jmp

create vectors-out
next 1lo $ lda, exitj sta,
coloni sta,
next hi $ lda, exitj sta,
columni sta,

w + 1+ lda, nextj sta,

push 1lo $ lda, dupjs sta,
push 1hi $ lda, dupns 1+ sta,

( extension to colonj )
```

```
code wexecute ( cfa -- )

wsave $ ssave $

nextjs $ dup : r 1+ indent! :

wsave $ dup .word cr wexecute
r dup 2+ 1save $ wsave !

r indent! :

r indent! :

r ssave !

status

else

"primitive"
then

w walk ( cfa -- )
dup walkable?
if

wexecuting dup .word cr wexecute
else

r cr . stack cr . word cr break r - execute

stack cr break
then

w walk ( -- )

"walking" dup .word cr wexecute
```

```
create morecoloni
nest# inl,
next jmp,

( extension to doesj )

create moreexit
nest# inl,

push jmp,

( extension to doesj )

create morenext ( addr -- )
nest# lda, or
if

( valid only for colas-forth )

( jmp vectors that are changed )

( *********************** )

( assembly language subroutines )

( *********************** )

create lo
 o o lda, i + i ....

create hi
 i + i lda, + i sta,
 + i lda jmp

create vectors-out
next 1lo $ lda, exitj sta,
```
MEMBERSHIP
IN THE FORTH INTEREST GROUP

107 - MEMBERSHIP in the FORTH INTEREST GROUP & Volume 7 of FORTH DIMENSIONS. No sales tax, handling fee or
discount on membership. See the back page of this order
form.

The FORTH INTEREST GROUP is a worldwide non-profit member-supported
organization with over 5,000 members and 80 chapters. FIG membership
includes a subscription to the bi-monthly publication, FORTH Dimensions.
FIG also offers its members publication discounts, group health and life
insurance, an on-line data base, a large selection of Forth literature, and
many other services. Cost is $20.00 per year for USA, Canada & Mexico;
all other countries may select surface ($27.00) or air ($33.00) delivery.

The annual membership dues are based on the membership year, which
runs from May 1 to April 30.

When you join, you will receive issues that have already been circulated
for the current volume of FORTH Dimensions and subsequent issues will be
mailed to you as they are published.

You will also receive a membership card and number which entitles you to
a 10% discount on publications from FIG. Your member number will be
required to receive the discount, so keep it handy.

HOW TO USE THIS FORM

1. Each item you wish to order lists three different Price categories:
   - Column 1 - USA, Canada, Mexico
   - Column 2 - Foreign Surface Mail
   - Column 3 - Foreign Air Mail

2. Select the item and note your price in the space provided.

3. After completing your selections enter your order on the fourth page of this form.

4. Detach the form and return it with your payment to The Forth Interest Group.

FORTH DIMENSIONS BACK VOLUMES
The six issues of the volume year (May - April)

101 - Volume 1 FORTH Dimensions (1979/80) $15/16/18
102 - Volume 2 FORTH Dimensions (1980/81) $15/16/18
103 - Volume 3 FORTH Dimensions (1981/82) $15/16/18
104 - Volume 4 FORTH Dimensions (1982/83) $15/16/18
105 - Volume 5 FORTH Dimensions (1983/84) $15/16/18
106 - Volume 6 FORTH Dimensions (1984/85) $15/16/18

ASSEMBLY LANGUAGE SOURCE CODE LISTINGS
Assembly Language Source Listings of fig-Forth for specific CPUs and
machines with compiler security and variable length names.

513 - 1802/MARCH 81 .................. $15/16/18

514 - 6502/SEPT 80 .................. $15/16/18
515 - 6800/MAY 79 .................. $15/16/18
516 - 6809/JUNE 80 .................. $15/16/18
517 - 8080/SEPT 79 .................. $15/16/18
518 - 8086/88/MARCH 81 .......... $15/16/18
519 - 9900/MARCH 81 ................ $15/16/18
520 - ALPHA MICRO/SEPT 80 ........ $15/16/18
521 - APPLE II/AUG 81 .............. $15/16/18
522 - ECLIPSE/OCT 82 .............. $15/16/18
523 - IBM-PC/MARCH 84 .............. $15/16/18
524 - NOVA/MAY 81 .................. $15/16/18
525 - PACE/MAY 79 .................. $15/16/18
526 - PDP-11/JAN 80 ................. $15/16/18
527 - VAX/OCT 82 .................. $15/16/18
528 - Z80/SEPT 82 .................. $15/16/18
### BOOKS ABOUT FORTH

<table>
<thead>
<tr>
<th>Title</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 - ALL ABOUT FORTH</td>
<td>$25/26/35</td>
</tr>
<tr>
<td>Glen B. Haydon</td>
<td></td>
</tr>
<tr>
<td>An annotated glossary for MVP Forth; a 79-Standard Forth.</td>
<td></td>
</tr>
<tr>
<td>205 - BEGINNING FORTH</td>
<td>$17/18/21</td>
</tr>
<tr>
<td>Paul Chirlian</td>
<td></td>
</tr>
<tr>
<td>Introductory text for 79-Standard.</td>
<td></td>
</tr>
<tr>
<td>215 - COMPLETE FORTH</td>
<td>$16/17/20</td>
</tr>
<tr>
<td>Alan Winfield</td>
<td></td>
</tr>
<tr>
<td>A comprehensive introduction including problems with answers. (Forth 79)</td>
<td></td>
</tr>
<tr>
<td>220 - FORTH ENCYCLOPEDIA</td>
<td>$25/26/35</td>
</tr>
<tr>
<td>Mitch Derick &amp; Linda Baker</td>
<td></td>
</tr>
<tr>
<td>A detailed look at each FIG-Forth instruction.</td>
<td></td>
</tr>
<tr>
<td>225 - FORTH FUNDAMENTALS, V. 1</td>
<td>$16/17/20</td>
</tr>
<tr>
<td>Kevin McCabe</td>
<td></td>
</tr>
<tr>
<td>A textbook approach to 79-Standard Forth.</td>
<td></td>
</tr>
<tr>
<td>230 - FORTH FUNDAMENTALS, V. 2</td>
<td>$13/14/16</td>
</tr>
<tr>
<td>Kevin McCabe</td>
<td></td>
</tr>
<tr>
<td>A glossary.</td>
<td></td>
</tr>
<tr>
<td>232 - FORTH NOTEBOOK</td>
<td>$25/26/35</td>
</tr>
<tr>
<td>Dr. C. H. Ting</td>
<td></td>
</tr>
<tr>
<td>Good examples and applications. Great learning aid. PolyFORTH is the dialect used. Some conversion advice is included. Code is well documented.</td>
<td></td>
</tr>
<tr>
<td>233 - FORTH TOOLS</td>
<td>$19/21/23</td>
</tr>
<tr>
<td>Gary Feierbach &amp; Paul Thomas</td>
<td></td>
</tr>
<tr>
<td>The standard tools required to create and debug Forth-based applications.</td>
<td></td>
</tr>
<tr>
<td>235 - INSIDE F-83</td>
<td>$25/26/35</td>
</tr>
<tr>
<td>Dr. C. H. Ting</td>
<td></td>
</tr>
<tr>
<td>Invaluable for those using F-83.</td>
<td></td>
</tr>
<tr>
<td>237 - LEARNING FORTH</td>
<td>$17/18/21</td>
</tr>
<tr>
<td>Margaret A. Armstrong</td>
<td></td>
</tr>
<tr>
<td>Interactive text, introduction to the basic concepts of Forth. Includes section on how to teach children Forth.</td>
<td></td>
</tr>
<tr>
<td>240 - MASTERING FORTH</td>
<td>$18/19/22</td>
</tr>
<tr>
<td>Anita Anderson &amp; Martin Tracy (MicroMotion)</td>
<td></td>
</tr>
<tr>
<td>A step-by-step tutorial including each of the commands of the Forth-83 International Standard; with utilities, extensions and numerous examples.</td>
<td></td>
</tr>
<tr>
<td>245 - STARTING FORTH (soft cover)</td>
<td>$20/21/24</td>
</tr>
<tr>
<td>Leo Brodie (FORTH, Inc.)</td>
<td></td>
</tr>
<tr>
<td>A lively and highly readable introduction with exercises.</td>
<td></td>
</tr>
<tr>
<td>246 - STARTING FORTH (hard cover)</td>
<td>$24/25/29</td>
</tr>
<tr>
<td>Leo Brodie (FORTH, Inc.)</td>
<td></td>
</tr>
<tr>
<td>255 - THINKING FORTH (soft cover)</td>
<td>$16/17/20</td>
</tr>
<tr>
<td>Leo Brodie</td>
<td></td>
</tr>
<tr>
<td>The sequel to “Starting Forth”. An intermediate text on style and form.</td>
<td></td>
</tr>
<tr>
<td>265 - THREADED INTERPRETIVE LANGUAGES</td>
<td>$23/25/28</td>
</tr>
<tr>
<td>R.G. Loeliger</td>
<td></td>
</tr>
<tr>
<td>Step-by-step development of a non-standard Z-80 Forth.</td>
<td></td>
</tr>
<tr>
<td>270 - UNDERSTANDING FORTH</td>
<td>$3.50/5/6</td>
</tr>
</tbody>
</table>
THE JOURNAL OF FORTH APPLICATION & RESEARCH
A refereed technical journal published by the Institute for Applied Forth Research, Inc.

401 - JOURNAL OF FORTH RESEARCH V.1 #1 $15/16/18
Robotics.

402 - JOURNAL OF FORTH RESEARCH V.1 #2 $15/16/18
Data Structures.

403 - JOURNAL OF FORTH RESEARCH V.2 #1 $15/16/18
Forth Machines.

404 - JOURNAL OF FORTH RESEARCH V.2 #2 $15/16/18
Real-Time Systems.

405 - JOURNAL OF FORTH RESEARCH V.2 #3 $15/16/18
Enhancing Forth.

406 - JOURNAL OF FORTH RESEARCH V.2 #4 $15/16/18
Extended Addressing.

407 - JOURNAL OF FORTH RESEARCH V.3 #1 $15/16/18
Forth-based laboratory systems and data structures.

REPRINTS
420 - BYTE REPRINTS $5/6/7
Eleven Forth articles and letters to the editor that have appeared in Byte magazine.

421 - POPULAR COMPUTING 9/83 $5/6/7
Special issue on various computer languages, with an in-depth article on Forth's history and evolution.

DR. DOBB'S JOURNAL
This magazine produces an annual special Forth issue which includes source-code listings for various Forth applications.

422 - DR. DOBB'S 9/82 $5/6/7

423 - DR. DOBB'S 9/83 $5/6/7

424 - DR. DOBB'S 9/84 $5/6/7

425 - DR. DOBB'S 10/85 $5/6/7

HISTORICAL DOCUMENTS
501 - KITT PEAK PRIMER $25/27/35
One of the first institutional books on Forth. Of historical interest.

502 - FIG-FORTH INSTALLATION MANUAL $15/16/18
Glossary model editor - We recommend you purchase this manual when purchasing the source-code listings.

503 - USING FORTH $20/21/23
FORTH, Inc.

REFERENCE
305 - FORTH 83 STANDARD $15/16/18
The authoritative description of 83-Standard Forth. For reference, not instruction.

300 - FORTH 79 STANDARD $15/16/18
The authoritative description of 79-Standard Forth. Of historical interest.

316 - BIBLIOGRAPHY OF FORTH REFERENCES
2nd edition, Sept. 1984 $15/16/18
An excellent source of references to articles about Forth throughout microcomputer literature. Over 1300 references.

MISCELLANEOUS
601 - T-SHIRT SIZE
Small, Medium, Large, and Extra-Large. White design on a dark blue shirt. $10/11/12

616 - HANDY REFERENCE CARD FREE

683 - FORTH-83 HANDY REFERENCE CARD FREE

FORTH INTEREST GROUP
WINTER SPECIAL
3 FOR $10.00
Dr. Dobb's Journal Annual Forth Issues
3 for $10.00 9/82, 9/83, 9/84
USA-Canada only $10/12/14 (9/82, 9/83, 9/84)

NEW SERVICE FROM THE FORTH INTEREST GROUP
The Forth Interest Group is now providing a free telephone job referral service for members and potential employers or contractors. The purpose is to provide contact between Forth programmers and persons or companies seeking their services.

Members of the Forth Interest Group may register with the Referral Service by completing the form below and returning it to:

FIG
P.O. Box 8231
San Jose, CA 95155

Prospective employers or clients may contact the FIG office by calling (408) 277-0668. We will be happy to provide contact information for FIG members who are registered with the referral service.

FIG JOB REFERRAL FORM
Member Number Date
Name
Address
Are you seeking:
Full-Time Employment Consulting Contracts
How many years programming in Forth professionally?
Total Years of Forth programming?
Familiar with:
Computer(s) - (Limit to 30 characters)
Version(s) of Forth - (Limit to 30 characters)

FIG Dimensions
<table>
<thead>
<tr>
<th>ITEM #</th>
<th>TITLE</th>
<th>AUTHOR</th>
<th>QTY</th>
<th>UNIT PRICE</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>107</td>
<td>MEMBERSHIP</td>
<td></td>
<td></td>
<td></td>
<td>SEE BELOW</td>
</tr>
</tbody>
</table>

☐ Check enclosed (payable to: FORTH INTEREST GROUP)
☐ VISA     ☐ MASTERCARD

Card # ____________________________
Expiration Date ____________________
Signature __________________________

<table>
<thead>
<tr>
<th>SUBTOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>10% MEMBER DISCOUNT</td>
</tr>
<tr>
<td>CA. RESIDENTS SALES TAX</td>
</tr>
<tr>
<td>HANDLING FEE</td>
</tr>
<tr>
<td>MEMBERSHIP FEE</td>
</tr>
<tr>
<td>$20/27/33</td>
</tr>
</tbody>
</table>

TOTAL

PAYMENT MUST ACCOMPANY ALL ORDERS

MAIL ORDERS
Send to:
FORTH INTEREST GROUP
P.O. Box 8231
San Jose, CA 95155

PHONE ORDERS
Call 408/277-0668 to place credit card orders or for customer service. Hours: Monday-Friday, 9am-5pm PST.

PRICES
All orders must be prepaid. Prices are subject to change without notice. Credit card orders will be sent and billed at current prices. $15 minimum on charge orders. Checks must be in US, drawn on a US bank. A $10 charge will be added for returned checks.

POSTAGE & HANDLING
Prices include shipping. A $2.00 handling fee is required with all orders.

SHIPPING TIME
Books in stock are shipped within five days of receipt of the order. Please allow 4-6 weeks for out-of-stock books (delivery in most cases will be much sooner).

SALES TAX
California deliveries add 8%. San Francisco Bay Area add 7%.
Code Modules and Data Structures

John S. James
Santa Cruz, California

This tutorial grew out of work on a memory management problem of the Forth component-library project (see "Forth Component Libraries," Forth Dimensions VII/4). This project has the goal of allowing easy exchange of Forth modules - large components of programs - among different development environments and different installations. In other words, the library system must support a market for off-the-shelf packaged modules, such as sorts, B-tree database access and network interfaces.

Several factors complicated the question of how these modules would access data memory:

(1) A reliable library system requires completely automatic loading of modules, with no tinkering at all by different developers, groups, or institutions - not even patching of constants to tell the module what memory is available. Otherwise, whoever made the modifications would risk introducing errors, and would need to understand internals of the modules in order to minimize this risk. In short, Forth would be where it is now, without effective off-the-shelf components, with most programs written from scratch and with large projects risking excessive complexity when there are thousands of words in the dictionary and few reliable boundaries between program components.

(2) For many reasons (including ROMability, multi-tasking and recursion), library modules should compile into pure code; they should not put variables into the dictionary.

(3) Often, the module's author and even the programmer who uses the module in an application won't know how much memory will be needed at run time.

Many programming languages face these requirements and provide memory management facilities to meet them. But Forth is weak in memory management. Programmers can implement anything they want, of course, but the Forth language has little built in. So programmers cannot rely on a standard way, available in all systems, to get and release memory as needed.

Memory for Modules

One possible approach would be to implement memory management as part of the library support system, and make all authors of modules use the calls provided. But it would be difficult or impossible to impose a single system without causing intolerable problems for a language and a community which highly value flexibility. Perhaps we should standardize memory management in Forth; but that issue should be handled separately from the question of whether to have a component library.

So instead of providing memory management, we implemented a simple facility to separate memory management from everything else in the library support system. A defining word, BVARIABLE (the B stands for "based"), allows modules to access memory through data structures which are relative to pointer values. These data structures, or "activation records," usually correspond to a current invocation of a module or to one instance of a data object.

With this approach, the library support system does not need to deal with memory management. Programmers calling the modules can kludge any memory management they want, using ALLOT, PAD, address constants, or whatever — outside of the modules. Or they can be more elegant and use a library module written just for memory management; we expect that various modules will become available to suit different requirements and tastes. The library system can provide memory management; but it isn't hardwired to any particular implementation.

Using BVARIABLE

B_VARIABLE is a defining word which, when executed, takes two arguments from the stack: the address of a pointer and an offset value. B_VARIABLE adds to the dictionary a word which, when executed, takes nothing from the stack and returns an address which is the sum of the contents of the pointer and the offset.

B_VARIABLE is much like the common Forth word USER, which creates "user variables" for multi-tasking. Unlike ordinary variables, each task has its own copy of the user variables. USER takes one argument, the variable's offset into the "user area" (activation record) of the task. The multi-tasking system itself maintains the pointer value (usually in a register) to which this offset is added to access the variable. The programmer does not control this value; and each task may have only one user area.

B_VARIABLE differs from USER in that the application programmer does control the pointer value, and can change it at any time. Therefore, many different instances of a data structure may be active simultaneously, even in the same task. Before using a word which applies to that data structure, the programmer makes sure the pointer contains the correct value to select the particular instance to be accessed.

The word B_VARIABLE is defined in the library support system, so it is always available. A programmer who writes a library module uses B_VARIABLE to set up the various variables and any other data areas in the activation record. Later, the application programmer who wants to use the module must first define a word which returns the address of a pointer, before compiling the module. The module's documentation gives the name to be used for this word. Usually this pointer will be defined as an ordinary variable or as a user variable, or as a constant pointing to some available RAM.

Before executing any words in the module, the application programmer
must set this pointer to the address of some available memory which will be used for the activation record. This activation record must be initialized, and usually the module contains a convenient initialization word.

Note that the application programmer, not the library system, is responsible for memory management.

How \texttt{B_VARIABLE} Works

First we will illustrate the definition in higher-level Forth, using \texttt{CREATE ... DOES >}. But for practical use, \texttt{B_VARIABLE} must be optimized by using \texttt{CODE} instead of \texttt{CREATE ... DOES >}. Speed is very important, because the library modules will use the words created by \texttt{B_VARIABLE} for all their access to data.

The higher-level definition, which should run on any Forth-83 system, is:

\begin{verbatim}
: B_VARIABLE \name ; aptr noffset -- \P Creates a based variable \Child: -- address
CREATE ,
\texttt{CODE} W INC W INC 0 [W]
AX MOV (Offset)
W INC W INC 0 [W]
W MOV 0 [W] AX ADD
1PUSH END-CODE
\end{verbatim}

When this word is executed, it creates the new word in the dictionary (with \texttt{CREATE}), and compiles the offset and pointer address into the dictionary also, in the parameter field of the created word (the two commas do this). Then it changes the definition of the created word so that it will execute the code after the \texttt{DOES >}.

When the created (or “child”) word is executed, the address of its parameter field is left on the stack before control is given to the code following \texttt{DOES >}. \texttt{DUP @} puts the offset value on the stack. \texttt{SWAP 2 +} @ puts the address of the pointer on top of it, deleting the original address of the parameter field in the process. + adds the two to get the final address.

A \texttt{CODE} version of \texttt{B_VARIABLE}, only for the 8086/8088 and only for F83 (not necessarily for other Forth-83 systems for the 8086/8088), is:

\begin{verbatim}
: B_VARIABLE \name ; aptr noffset -- 
\texttt{P} Creates a based
\texttt{variable} \Child: -- address
CREATE ,
\texttt{CODE} W INC W INC 0 [W]
AX MOV (Offset)
W INC W INC 0 [W]
W MOV 0 [W] AX ADD
1PUSH END-CODE
\end{verbatim}

A full explanation of this definition would divert us into the internals of this particular Forth system. But we might mention that \texttt{W} is the name of a register used by the system to point to the word being executed; it’s okay to destroy its value within a code routine, as we have done here by using \texttt{W} as a scratch register which does not need to be saved and restored.

Here is a sample session which illustrates the use of \texttt{B_VARIABLE} (either version, as they

\texttt{VARIABLE PTR}
\texttt{PTR 0 B_VARIABLE X}
\texttt{PTR 15 B_VARIABLE Y}
\texttt{20000 PTR !}
\texttt{X 20000}
\texttt{Y 20015}
\texttt{25000 PTR !}
\texttt{X 25000}
\texttt{Y 25015}
\texttt{T 30000 0 DO X DROP LOOP ;}

The crude benchmark in the last line shows the speed improvement with the \texttt{CODE} version. An ordinary variable, a \texttt{B_VARIABLE} with the \texttt{CODE} definition, and a \texttt{B_VARIABLE} with the \texttt{CREATE} \texttt{DOES >} definition, were used as the \texttt{X} in the loop. The times were 2.5, 3.0 and 9.5 seconds; actual ratios are somewhat larger because of the constant time taken to execute the \texttt{DROP} and to loop back. Clearly, the \texttt{CODE} version would be recommended for practical use.

---

\textbf{NEW PRODUCTS}
- MODEL List Processor $40
  (with ELIZA and micro-LISP)
- MICROMOTION
  8276 S. Sepulveda BL #A171
  Los Angeles, CA 90045
A Universal Stack Word

Donie Hoekman
Santa Clara, California

Normally, good Forth code is characterized by an uncomplicated parameter stack. Sometimes, though, it’s difficult to avoid — such as when dealing with mixed-mode arithmetic, floating-point, quad-precision or graphics routines. Presented here is a single function called STACK which, by itself, can replace any stack manipulation word or any consecutive sequence of stack manipulation words. Basically, it allows you to do a lot of messy things on the stack all in one breath, without worrying about how things got there where you wanted them to go.

This trick is accomplished by providing the STACK word with a before-and-after representation of the stack. Here are some examples of how STACK would be used to implement some common Forth stack manipulation operators:

1. DROP STACK A] ;
2. DUP STACK A|AA ;
3. SWAP STACK AB|BA ;
4. ROT STACK ABC|BCA ;
5. 2SWAP STACK ABCD|CDAB ;

The characters to the left of | represent the input stack picture, and to the right is the output stack picture. The following limitations apply to STACK as it is implemented here:

(1) If the input stack consists of n items, the input stack picture must contain n consecutive ASCII characters, starting with A. This will normally limit the input stack depth to twenty-six.

(2) The output stack picture must contain only those characters found in the input stack picture. The output stack is limited to a depth of 127 characters.

Screen 2 contains the compile-time behavior of STACK. Screen 1 contains two versions of the run-time code.
ATTENTION FIG MEMBERS!
WE NEED YOUR HELP

At the FORTH Interest Group we know Forth is being used in many sophisticated and complicated projects. Unfortunately, the Forth community has never compiled a complete reference document summarizing how and where Forth is being used. We believe this type of document would be very helpful to both the novice considering learning Forth and the professional experiencing corporate resistance to using it.

Would you please help us put one together? All you need to do is complete the questionnaire below and return it directly to us by March 15! All completed questionnaires should be mailed to: Forth Interest Group, P.O. Box 8231, San Jose, CA 95155.

1. Company name and address:

2. Name of the programmer
   (Note: for internal use only. Will not be published.)

3. Project or product name

4. Date project or product completed

5. Was the project: For sale to an end user? yes no
   For in-house use? yes no
   For OEMs? yes no

6. Indicate approximate number of users:
   1–50
   50–100
   100–200
   200–300

7. Is Forth hidden from the user? yes no

8. Briefly describe the project (30 words)

9. Briefly describe the benefits of using this project or product.

Thank you for your participation. If you would like a copy of the results please complete the following.

Name

Company

Address

City, State, Zip

Most applications of STACK would require an assembly language implementation, but for experimentation a Forth implementation is also shown. However, the assembly implementation runs about ten times faster on an 8088–based system.

The assembly implementation of STACK compiles into about 200 bytes of code, while the Forth–83 version takes about 240 bytes. The assembly code for (STACK) should run on most 8088/8086 Forth systems, and the Forth (STACK) word should work on any standard Forth–83 system.

At compile time, STACK first lays down the run-time address of (STACK). It then counts the number of input and output items and compiles a single byte for each of these values. STACK then compiles one byte for each of the characters in the output stack picture. At run time, (STACK) moves each input stack item into temporary storage at HERE. (STACK) then runs through the output stack picture and recalls each item from the HERE buffer and pushes it onto the stack. To simplify the run-time code, the output stack items are stored in a special format. If there are n items on the input stack, then item m will be stored as (n–m)*2. This value then represents the offset address from HERE required to pick up the next output stack item. The total compile size of any STACK word will always be four bytes plus a byte for each item on the output stack.

Screen 3 shows some examples of using STACK versus doing it the “hard” way. The box words draw a box when provided with upper-left and lower-right coordinates. The star words draw a star when given five vertices. In each case, the STACK implementation required less object code, less source code, ran slightly faster on a repetitive benchmark and took much less time to get running.

To summarize, STACK should be used primarily to simplify the handling of complex stack pictures. For relatively simple manipulations, STACK will extract a performance penalty. However, when a programmer feels he has painted himself into a corner, STACK may be a good way to spell relief.
Fast Evaluation of Polynomials

Nathaniel Grossman
Los Angeles, California

One of the speakers at the Sixth
FORML Conference presented a clever utility that automatically parses a stack diagram and an algebraic formula, and generates explicit stack operations that realize the formula with the given stack parameters as ordered. Thus, the utility responds to the stack diagram C A B X and the algebraic formula AX^2 + BX + C by producing the sequence of Forth words \texttt{ROT OVER DUP * * ROT ROT * ROT + +}. When the stack diagram is changed to B C A X, say, the same formula is realized by a different sequence of Forth words. It is possible that one of the sequences produced by the permutations on the stack will be shorter than the others, and will be so found without requiring the programmer to go through the usual mental acrobatics or paper-and-pencil calisthenics.

Nevertheless, I was surprised to note that none of the other FORML participants seemed to see or care that no one of these permutations can produce a realization of the shortest possible evaluation of a quadratic polynomial:

\[ AX^2 + BX + C = (AX + B)X + C \]

While direct evaluation of the quadratic polynomial by the Forth word sequence given above requires three multiplications, the last expression gives the value with only two multiplications:

\texttt{QUADRATIC ( c b a x -- ax^2 + bx + c) SWAP OVER * ROT + + ;}

Because the two versions each use just two additions, the second version requires only about two-thirds the running time of the first version. One of the major themes running through the conference presentations was the striving for savings of one or two microseconds by painstaking reworking of code. Why be penny-wise and pound-foolish?

The rearrangement of the quadratic polynomial is the simplest case of a general scheme devised by the English mathematician W.G. Horner and published in 1819\(^5\). Horner’s scheme used to be taught in the second year of high school algebra, along with a simple algorithm for digit-by-digit extraction of roots of polynomials. The appearance of electronic arithmetic engines made the use of digit-by-digit methods less a necessity, and Horner’s method seems now to be in eclipse, at least to the extent that few high school students carry knowledge of the method forward into college studies. Of course, Horner’s scheme has not been lost. Practitioners of numerical analysis prize Horner’s scheme because it is the fastest of all ways to evaluate polynomials generically: if the polynomial has no special features, no algorithm for evaluating the polynomial can use fewer multiplications than Horner’s scheme\(^9\). If the polynomials do have special features, there may very well be algorithms faster than Horner’s scheme for that particular polynomial. For example, the simple polynomial \(x^n\) needs \(n-1\) multiplications for evaluation by Horner’s scheme, but there are methods that evaluate \(x^n\) in about \(\log_2 n\) multiplications. (The direct evaluation of a generic \(n\)th degree polynomial by working out all products will require \(1 + 2 + \ldots + n = n(n+1)/2\) multiplications.)

We will present Forth code to implement Horner’s scheme both in the form already described and in an extended form that evaluates the derivative polynomial simultaneously. With these two values available, we easily can employ the famous Newton’s method for finding roots of polynomials.

\textbf{Horner’s Scheme}

Horner’s scheme is contained in the algebraic identity

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_0 = \left( \ldots (a_n x + a_{n-1}) x + a_{n-2} x \ldots + a_1) x + a_0 \right. \]

The expression on the right side is evaluated from the inside out. The calculation can be set out in algorithmic form\(^2\).\(^9\). Let \(b_0 = a_0\) and compute \(b_{n-1}, b_{n-2}, \ldots, b_1, b_0\) by the recurrence \(b_k = a_k + b_{k+1} \times x_0\). Then \(b_0 = p(x_0)\). In addition, set

\[ q(x) = b_n x^{n-1} + b_{n-1} x^{n-2} + \ldots + b_2 x + b_1 \]

Then \(p(x) = (x - x_0) q(x) + b_0\) and the derivative \(p' (x) = q(x_0)\). The value of \(q(x_0)\) can be found by applying Horner’s scheme to \(q(x)\). Thus, the data is degree \(n\); coefficients \(a_0, a_1, \ldots, a_n; x_0\). Put \(y_0 = a_n\) and \(z_0 = a_n\). Compute successively the quantities

\[ y_k = x_0 \times y_{k-1} + a_k\]

and \(z_k = x_0 \times y_{k-1} + z_{k-1}\) for \(k = n-1, n-2, \ldots, 1\), and also \(y_0\). Then \(p(x_0) = y_0\) and \(p'(x_0) = z_1\).

The Forth screens that accompany this text hold words \texttt{HORNER\_SCHEME} and \texttt{FHORNER\_SCHEME} to realize the Horner polynomial evaluation schemes in integer and floating-point arithmetic, respectively, and a word \texttt{NEWTON\_CYCLE} including evaluation of a polynomial and its derivative simultaneously in floating-point arithmetic.

\textbf{Newton’s Method for Polynomials}

There are times when one wants a root of the polynomial \(p(x)\), a real number \(r\) such that \(p(r) = 0\). Indeed, the finding of roots of polynomials is the fundamental problem of classical algebra, and it is nowhere near being solved in the form of a “sure-fire” algorithm. However, numerical analysts can call upon an arsenal of techniques for locating roots and calculating them to a desired precision. Newton’s method is one of the most powerful of such techniques. If a root is located approximately, Newton’s method generates successive approximations that converge very rapidly to the root in most cases.

Newton’s method is expressed in a very simple form. Suppose that \(f(x)\) is a differentiable function in a “suitable” interval containing the root \(r\) of \(f(x) = 0\). Choose a first guess \(x_0\) for the root \(r\). (This is the hard part, because no turn-the-crank techniques are known for this step that are both “sure-fire” and rapid.) Then generate a sequence of improved approximations \(x_1, x_2, \ldots\) by the iteration scheme

\[ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \]
If certain technical conditions are met, the sequence \( x_n \) will converge rapidly to the root \( r \), and the convergence is quadratic: eventually, the number of correct decimal places in the approximation to the root \( r \) will essentially double with each step.

Of course, a polynomial is a function differentiable everywhere and so Newton's method can be used to approximate roots of polynomials. There are some pitfalls, naturally, so potential users should glance at a numerical analysis textbook such as that written by Burden to become aware of them. Forth devotees will be sensitive to the question of calculation in fixed-point arithmetic. They will find a possibly useful discussion of Newton's method and fixed-point calculations in an earlier paper. The implementation presented here will be in floating-point arithmetic.
SCR # 2
0 \ Poly-array
1 2 : POLY_ARRAY
2 CREATE , ( n --- )
3 \ The degree n will be followed by the n+1 integer coefficients
4 DOES> ( n --- nth coeff if n := 0, degree if n < 0 )
5 SWAP DUP @ ( degree )
6 ELSE 2* 2+ @ ( nth coefficient )
7 THEN ;
8 DEFER COEFFICIENT
9
SCR # 3
0 \ Horner_scheme
1 2 : HORNER_SCHEME ( n --- n1 = p[n] )
2 -1 COEFFICIENT ( degree) DUP COEFFICIENT ( leading coeff)
3 SWAP 1- 0 SWAP DO
4 OVER * 1 COEFFICIENT +
5 -1 +LOOP ;
6
SCR # 4
0 \ Examples
1 2 3 POLY_ARRAY PQ
2 -6 , 11 , -6 , 1 ,
3 4
5 : PQ() ( n --- n1 = pq[n] )
6 ['] PQ IS COEFFICIENT
7 HORNER_SCHEME ;
8
9 6 POLY_ARRAY RS
10 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 , 1 ,
11
12 : RS() ( n --- n1 = rs[n] )
13 ['] RS IS COEFFICIENT
14 HORNER_SCHEME ;
15
NGS FORTH
A FAST FORTH,
OPTIMIZED FOR THE IBM
PERSONAL COMPUTER AND
MS-DOS COMPATIBLES.

STANDARD FEATURES
INCLUDE:

- 79 STANDARD
- DIRECT I/O ACCESS
- FULL ACCESS TO MS-DOS FILES AND FUNCTIONS
- ENVIRONMENT SAVE & LOAD
- MULTI-SEGMENTED FOR LARGE APPLICATIONS
- EXTENDED ADDRESSING
- MEMORY ALLOCATION CONFIGURABLE ON-LINE
- AUTO LOAD SCREEN BOOT
- LINE & SCREEN EDITORS
- DECOMPILER AND DEBUGGING AIDS
- 8088 ASSEMBLER
- GRAPHICS & SOUND
- NGS ENHANCEMENTS
- DETAILED MANUAL
- INEXPENSIVE UPGRADES
- NGS USER NEWSLETTER

A COMPLETE FORTH
DEVELOPMENT SYSTEM.

PRICES START AT $70

NEW◆HP-150 & HP-110
VERSIONS AVAILABLE

NEXT GENERATION SYSTEMS
P.O. BOX 2987
SANTA CLARA, CA. 95055
(408) 241-5909
When the function $f(x)$ is a polynomial, the Horner scheme is very convenient for organizing the calculations in the Newton iteration.

The Forth Screens

The accompanying Forth screens implement the algorithms just laid out. They are written in Forth–83 and I have checked them with MicroMotion MasterForth, including the floating-point extension, for which a proposed standard can be obtained. Screen 0 is merely descriptive and cannot be loaded. Screen 1 is the loader screen.

I decided to write a generic program. Therefore, I have to defer entry of the coefficient arrays and I use for that purpose the two vectoring words DEFER and IS whose definitions, taken from Mastering Forth, are shown in Figure One.

The coefficients of the polynomial to be evaluated are saved in an array of floating-point numbers. We may want to deal with more than one polynomial at once and, therefore, are led to two new data types. The POLY_FARRAY prefices the list of integer coefficients by a sixteen-bit integer that is the degree of the polynomial. This integer must be at hand for indexing certain DO LOOPS. The POLY_FARRAY is similar, except that the coefficients stored are floating-point numbers.
Screen 2 Here is the first of the new data types. As illustrated on screen 4, the defining word is used in the form

\[ \text{<degree>} \text{ POLY\_ARRAY <name>} \]

to create an array that enters the dictionary as the string of cells

\[ | \text{deg} | a_0 | a_1 | \ldots | a_{n-1} | a_n | \]

holding the degree followed by the coefficients of the polynomial \(<name>\).

Screen 3 The deferred word \text{FCOEFFICIENT} in screen 2 will be the target for the vectored arrays of coefficients. Its immediate use is to allow writing the word \text{HORNER\_SCHEME} that carries out efficient evaluation of integer-coefficient polynomials for integer values of the variable.

Screen 4 These examples show how evaluation of any polynomial can be vectored through the word \text{HORNER\_SCHEME}. The first is the polynomial

\[ pq(x) = x^3 - 6x^2 + 11x - 6 = (x - 1)(x - 2)(x - 3) \]
The second is

\[ rs(x) = 1 + x + x^2 + \ldots + x^6 = \frac{(x^7 - 1)}{(x - 1)} \]

**Screen 5** This screen and the next two are parallel to screens 2 – 4. The floating-point words are taken from the proposed floating-point standard published by Duncan and Tracy. The word \( \text{F}_{i} \) is analogous to the integer word \( \text{F}_{i} \), and is not included in the proposed standard. The words \( \text{POLY\_ARRAY} \) and \( \text{FCOEFFICIENT} \) are completely analogous to the earlier words \( \text{POLY\_ARRAY} \) and \( \text{FCOEFFICIENT} \).
Screen 7  These polynomials are handled in complete analogy to those in screen 4.

Screen 8  The word Newton_cycle carries out a single Newton iteration step. Lines 4 and 5 contain the little fudge that is necessary because the stack contains mixed number types. The evaluations of p(x) and p'(x) are carried out in parallel on the stack, and the definition is left unfactored to ease the following of stack manipulations. Yes, I know there is a movement toward the introduction of more local variables with the consequent shorter stacks, but I could not resist the challenge of manipulating four-deep stacks without the aid of the phantom words Fpick and Froll. Just after the Do, the stack holds x y x z x.

Screen 9  To ensure that the iterations do not run for infinitely many cycles, two stoppers are set. Each iterate is compared with the preceding and the iterations are stopped if they differ in absolute value by less than the preset TOLERANCE. Because Newton’s method soon begins to double the number of correct decimal places at each iteration, the preset tolerance should soon stop the process. Just in case the initial guess is terrifically far from a root, we solve an equation treated by Vieta in the year 1600. Vieta had an iterative method, used and simplified by several mathematicians afterward until it was taken up and again refined by Newton in 16645. The equation is the cubic

\[ x^3 + 30x - 14356197 = 0 \]

Vieta started by guessing for the root the value \( x_0 = 200 \). We type in the following words:

```
3 POLY_FARARRAY VRAYA
-14356197F, 30F, 0F, 1F,
' VIETA IS FCOEFFICIENT
```

We want both the root and a look at the number of iterations used, so we continue with

```
200F NEWTON'S_S.CHEME F.
10 #CYCLES @ - .
```
The display reads
243.000000000 5

We have found the root 243 also found by Vieta, and the process consumed five iterations. Incidentally, there is no other real number root for this equation, as easily can be shown with a little bit of differential calculus.

References
The annual conference of the Forth Modification Laboratory (FORML) was held in Asilomar's Chapel this year, a lofty structure of raftered ceilings, wooden beams and comfortable seating. It is perched on a hill overlooking cloudbanks as they beach themselves on the Pacific shoreline. Most of the eighty or more participants were Forth experts who contributed their expertise to the schedule of talks, poster sessions and working groups. Complete papers will soon be available in the published proceedings, but the following material sketches some of the highlights in brief.

Dana Redington spoke on his work with knowledge representation. Some of his earlier techniques were revised recently when Dave Boulton, one of the founding members of FIG, helped Dana to reconsider what a "fact" is, enabling Dana to further refine his expert programs. His fact structure consists of a subject, verb, object and "certainty" (which provides a measure of the likelihood of correctness). Dana demonstrated his rule compiler with simple examples and shared with the audience his insight into expert systems and knowledge representation in Forth.

Jack Park next discussed his method of parsing natural languages. His explanation began with semantic and syntactic parsing, and with the way his "expectation parser" uses a Forth dictionary as a kind of smart vocabulary to discern between multiple possible word meanings. Jack's talk made it clear that the techniques used and the choice of language enable one to easily create or customize rules for how words in the dictionary are related.

Forth Inc.'s Jon Waterman reported a use of mailboxes to facilitate inter-task communication in a complex multitasking situation. Shared memory provides a place for access to data by both master (68000) and slave (8031) processors communicating at 375 kilobaud. The mailbox provides parameters for each of the communication channels. As with most discussions of multitasking, emphasis was given to communication and control protocols.

An entire session of this year's conference was devoted to languages. Beginning the late afternoon session was Martin Tracy, of MicroMotion, who discussed adding LISP operators to Forth. The paper on which the talk was based includes much source code and pertinent information about list handlers. To demonstrate the effectiveness of the resulting Forth/LISP list handler, Martin has used it to implement Winston and Horn's "micro-LISP" (LISP, 2nd ed.). Martin concluded that LISP is a good idea, but takes its own premises too far when it insists that everything in it must take the form of a list.

Lance Collins travelled from Victoria, Australia to describe implementing a LOGO compiler in Forth. Of particular interest was his use of stack frames and frame pointers to resolve the problem of local variables in recursive definitions. His paper explains one company's approach to providing local variables, forward referencing and causing any redefined word to affect all previous references to it. A critical goal of the project was to allow standard LOGO programs to run unchanged on this system. Lance points out that LOGO is much more complex than it needs to be, when one considers the prevalent current uses of the language.

Included in the conference proceedings is a paper by Dr. C.H. Ting, who has implemented Prolog in Forth. He shows that since Prolog only requires a pointer to the query string and another into the data base, implementing it does not require new data structures in Forth. (George Levy was another speaker who shared with attendees his good work implementing Prolog in Forth.) Dr. Ting used his speaking time to describe his work (called "Footsteps in an Empty Valley") putting together a low-cost board based on the NC-4000 Forth processor. He spoke of the Novix chip as a milestone, and suggested that work is now needed on writing the systems utilities and controllers needed to harness its power.

Leonard Morgenstern has explored a method of accepting complex input and analyzing it effectively with his BNF parser written in Forth. This parser can handle serial, parallel and repetitive structures, or a combination of the three. The method makes it fairly simple to quickly write a parser called for by a given situation. The presentation left open many intriguing ideas for future exploration; many who attended will be looking forward to hearing about further development of this tool.

Wil Baden next presented his understanding of interpretive logic. One of his wishes was to use IF ELSE THEN for conditional compiling. Conditional interpretation and conditional compilation provide a number of benefits in one's normal work with Forth, especially by extending the power of the Forth editor and by using interpretive tests to interact with the compiling process. His ideas are fertile ground for the imagination.

David Harralson spoke on extending Forth's control structures to meet the programming requirements of the 1990s. His goal was to define a most general control structure with a minimum of new words, and to incorporate in it the current and proposed features of Forth's control structures. The scope of the task: that the new structure will run existing code with no speed penalty, while providing significant new capabilities and remaining flexible.

Martin Worrell of Hull, England needed a metacompiler to compile code for the MetaForth machine's development. The team wanted an "open metacompiler" which would allow one to test the metacompilation process interactively while it happens, permitting revision and replacement of individual pieces of code. The concepts of resolution and relocation become critical to this technique, which does not apply to cross-compilation. For most systems, the sophistication would come at a hefty cost in speed, but because this metacompiler was designed for the 10 mips MetaForth machine, the cost is very affordable.

Larry Forsley, of the Institute for Forth Application and Research and the University of Rochester, started off
the first evening with his observations gained from teaching people about Forth. He has given Forth seminars for the ACM, IEEE and other organizations. Larry perceives that different groups in need of instruction will require different approaches, and that the most successful Forth instructor will be the one who is most sensitive to the best approach for a given situation. One needs to motivate class members, understand one’s own purpose and, above all, the instructor must convey a unifying paradigm for the hardware, software and real world.

The better part of this evening session was devoted to coding style and conventions. Wil Baden reviewed his method for formatting phrases, and described his decompiler that generates formatted code. Kim Harris, a founding member and Secretary of FIG, then revealed the results of an informal survey conducted last year about Forth programmers’ preferences for coding conventions. On many of the points there was a consensus of opinion, but on other issues (such as a standard date format, what constitutes a “phrase” of Forth words, and indentation) the variety of opinion would suggest that a common conclusion remains elusive. Anyone who has managed or participated in large-scale development with a group of programmers can appreciate the need for common conventions to ease code sharing and maintenance. Kim also presented his automatic structure chart used for analysis of Forth programs. This important tool was very well received by the FORML group and will be receiving more attention as awareness of it spreads.

The theme of this year’s FORML conference was “Software Tools,” and the sessions devoted to it were of great interest. Loring Craymer began with his portability wordset. The objective was to write a wordset which would permit the easy transport of code between sixteen-bit and thirty-two-bit implementations. His tool deals with the problems of cell and token sizes, return stack access and virtual memory addressing.

“Self-Understanding Programs” were described by Mitch Bradley. He provided words that can unravel compiled colon definitions, regardless of the format in which the implementor has chosen to store the bodies of the words. This technique makes possible the creation of largely portable decoders, for example, that can also deal sensibly with user-defined compiling words and their progeny.

Working groups comprised a popular part of this FORML conference. People gathered for several hours to work on issues related to robotics, education, floating point, control structures, files, thirty-two-bit implementations, software tools and the NC-4000 chip. As can be expected, some of the groups reported consensus or useful conclusions, while others provided an opportunity to further explore individual viewpoints and diverse proposals. Each group’s moderator reported orally to an enthusiastic reconvened audience.

The topic of education was explored primarily by attendees who are involved in academic settings, including instructors who teach Forth at Stanford University and at UCLA. Larry Forsley related that the lack of a good, academically oriented textbook was seen as a major problem in reaching universities and colleges. Another desirable educational tool is a book of Forth solutions or case studies/algorithmic viewpoints to which particularly useful.

Robert Smith reported on the deliberations about floating point. It was generally agreed that floating point should be included as an extension to the standard, since vendors will supply it anyway. Standard numeric I/O should be required, it was felt, and a double-number format should be specified. Among potentially useful tools, a primitive that can divide by a double number was found desirable. It was undecided whether a separate floating-point stack should be implemented or whether the return stack should be used. Finally, participants recommended that a Forth Standards Team subcommittee be established to consider proposals for a common floating-point word set.

Control structures were discussed by a group moderated by David Harralson. The lively discussion was divided as to the adequacy of present control structures. The debate then turned to whether new proposals should be constrained to those which are completely compatible with Forth’s existing structures. Future work is planned along divergent lines of thought, which should result in some interesting new proposals in coming months.

Mitch Bradley led a group discussion about files. Several of those present had been using file-based systems and could agree on a standard set of word names for the interface. One solution was to write code that will read and write MS-DOS files from Forth (because MS-DOS tends to be nearly omnipresent, despite informed opinions as to the inefficiency of its file system), something that has been commercially implemented by at least one company.

The thirty-two bit discussions were reported by John Hart. One attendee argued that perfect backward compatibility with sixteen-bit code would not be achievable with available effort in a realistic period of time, considering the speed at which prevailing technology is evolving. “Gray matter conversion” was seen as the only practical solution to this aspect of the transportability dilemma. Considering the amount of memory typically available in today’s machines, the question is not seen as how to better use 64K, but how to get control of all available memory. The market is nearly demanding a solution to this problem.

Software tools were debated by a group led by Kim Harris. The types of needed tools fell into broad categories (e.g., design, management and prototyping) to which particularly useful tools can be ascribed; stack and control-flow analyzers, pretty printers, factoring analysis, on-line documentation, coding conventions and version control were notable examples.
Howard Johnson reported that the NC-4000 round table concentrated on the formation of a users group for that processor. The group is interested in bringing people together to share information about current projects using that Forth chip. The need to provide an environment for the chip was again brought up, and a number of people will need to get involved in creating one. A related meeting was scheduled for January 1986 in northern California.

On Saturday evening, a perennially popular session was held, consisting of five-minute impromptu talks. Martin Tracy spoke briefly about a Forth model library he is developing, which consists of large, substantial and well-documented programs that work. The library is intended to work on a wide variety of Forths under MS-DOS, and disks sell for $40 each.

A Forth system written in C was described by Mitch Bradley. His system is quite compatible with a Forth written in assembler. An interesting outcome is that this Forth can now be — and has been — ported to some of the powerful minicomputers for which there is not likely to be a specific Forth implementation but which do run C. Mitch concluded by offering to share his code with others.

Ray Gardner of Victoria, Australia presented some suggestions for work that would be well received in the machine vision industry. He explained some of the industry needs in physics, statistics, artificial intelligence and mathematics. Current machines tend to be quite costly in terms of the delivered results, revealing a market opportunity for those who are working, for example, with Forth hardware.

Many other impromptu speakers shared their insight — and wit — throughout the evening, which was capped by wine and cheese for all and informal chats through the late night hours. Speakers resumed their oral presentation of formal papers on the last morning of the conference.

Loring Craymer discussed improved error handling. He uses a stack frame to store a copy of stack items that can be restored after an error condition, rather than simply aborting. This allows one to get an error message from the system, which then continues running.

George Levy interested attendees with his simple expert system that learns through use (via interaction with the user) or by compiling a data base. The classic “Animals” game was George’s example, demonstrating the ease with which a rule compiler can be implemented in Forth. He was one of several presenters who emphasized Forth’s practicality for artificial intelligence. Jack Park, author of Expert II, noted that Levy’s technique is very clean, easily expanded with relatively trivial effort into non-trivial dimensions.

An ad hoc committee met to select two speakers whose work was worthy of special mention. Kim Harris’ automatic structure chart and Jack Park’s paper on parsing received awards, and Wil Baden and Leonard Morgenstern received honorable mentions, both for their work with parsers.

—Marlin Ouverson

Advertiser’s Index

Bryte 9
Dash, Find & Associates 13
Forth Inc. 10
Forth Institute 11
Forth Interest Group 19-22, 26, 40
Harvard Soft Works 15
HiTech Equipment 17
Laboratory Microsystems 8
MCA 15
MicroMotion 24
Miller Microcomputer Services 37
Mountain View Press 28
Next Generation Systems 29
Palo Alto Shipping Co. 2
Paradise Systems 4
Software Composers 34
SOTA 30
Talbot Microsystems 31
UBZ Software 32
<table>
<thead>
<tr>
<th>State</th>
<th>Chapter Name</th>
<th>City</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALABAMA</td>
<td>Huntsville FIG Chapter</td>
<td>Huntsville</td>
<td>Call Tom Konantz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>205/881-6483</td>
</tr>
<tr>
<td>ALASKA</td>
<td>mbKodiak Area Chapter</td>
<td>Kodiak</td>
<td>Call Horace Simmons</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>205/881-6483</td>
</tr>
<tr>
<td>ARIZONA</td>
<td>Phoenix Chapter</td>
<td>Phoenix</td>
<td>Call Dennis L. Wilson</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>602/956-7676</td>
</tr>
<tr>
<td></td>
<td>Tucson Chapter</td>
<td>Tucson</td>
<td>Twice Monthly, 2nd, &amp; 4th Sun., 2 p.m. Flexible Hybrid Systems 2030 E. Broadway #206 Call John C. Mead 602/323-9763</td>
</tr>
<tr>
<td></td>
<td>Central Arkansas Chapter</td>
<td>Arkansas</td>
<td>Twice Monthly, 2nd Sat., 2p.m. &amp; 4th Wed., 7 p.m. Call Gary Smith 501/227-7817</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monterey/Salinas Chapter</td>
<td>Monterey</td>
<td>Call Bud Devins 408/633-3253</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
|                  | Orange County Chapter        | Orange County               | 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 714/842-3023 San Diego Chapter Weekly, Thurs., 12 noon Call Guy Kelly 619/268-3100 ext. 4784 Sacramento Chapter Monthly, 4th Wed., 7 p.m. 1796-59th St., Room A Call Tom Ghormley 916/444-7775 Bay Area Chapter Silicon Valley Chapter Monthly, 4th Sat. FORML 10 a.m., Fig 1 p.m. ABC Christian School Aud. Dartmouth & San Carlos Ave. San Carlos Call John Hall 415/532-1115 or call the FIG Hotline: 408/277-0668 Stockton Chapter Call Doug Dillon 209/931-2448 COLORADO Denver Chapter Monthly, 1st Mon., 7 p.m. Call Steven Sarns 303/477-5955 CONNECTICUT Central Connecticut Chapter Call Charles Krajewski 203/344-9996 FLORIDA Orlando Chapter Every two weeks, Wed., 8 p.m. Call Herman B. Gibson 305/855-4790 Southeast Florida Chapter Monthly, Thurs., p.m. Coconut Grove area Call John Forsberg 305/252-0108 Tampa Bay Chapter Monthly, 1st Wed., p.m. Call Terry McNay 813/725-1245 GEORGIA Atlanta Chapter 3rd Tuesday each month, 6:30 p.m. Computone Cottillon Road Call Ron Skelton 404/393-8764 ILLINOIS Cache Forth Chapter Call Clyde W. Phillips, Jr. Oak Park 312/386-3147 Central Illinois Chapter Urbana Call Sidney Bowhill 217/333-4150 Fox Valley Chapter Call Samuel J. Cook 312/379-3242 Rockwell Chicago Chapter Call Gerard Kusiolek 312/885-8092 INDIANA Central Indiana Chapter Monthly, 3rd Sat., 10 a.m. Call John Oglesby 317/353-3929 Fort Wayne Chapter Monthly, 2nd Wed., 7 p.m. Indiana/Purdue Univ. Campus Rm. B71, Neff Hall Call Blair MacDermid 219/749-2042 IOWA Iowa City Chapter Monthly, 4th Tues. Engineering Bldg., Rm. 2128 University of Iowa Call Robert Benedict 319/337-7853 Central Iowa FIG Chapter Call Rodrick A. Eldridge 515/294-5659 Fairfield FIG Chapter Monthly, 4th day, 8:15 p.m. Call Gurdy Leete 515/472-7077 KANSAS Wichita Chapter (FIGPAC) Monthly, 3rd Wed., 7 p.m. Wilbur E. Walker Co. 532 Market Wichita, KS Call Arne Flores 316/267-8852 LOUISIANA New Orleans Chapter Call Darryl C. Olivier 504/899-8922 MASSACHUSETTS Boston Chapter Monthly, 1st Wed. Mitre Corp. Cafeteria Bedford, MA Call Bob Demrow 617/688-5661 after 7 p.m. MICHIGAN Detroit Chapter Monthly, 4th Wed. Call Tom Chrapkiewicz 313/562-8506 MINNESOTA MNFIG Chapter Even Month, 1st Mon., 7:30 p.m. Odd Month, 1st Sat., 9:30 a.m. Vincent Hall Univ. of MN Minneapolis, MN Call Fred Olson 612/588-9532 MISSOURI Kansas City Chapter Monthly, 4th Tues., 7 p.m. Midwest Research Institute MAG Conference Center Call Linus Orth 913/236-9189 St. Louis Chapter Monthly, 1st Tues., 7 p.m. Thornhill Branch Library Contact Robert Washam 91 Weis Dr. Ellisville, MO 63011 NEVADA Southern Nevada Chapter Call Gerald Hasty 702/452-3368 NEW HAMPSHIRE New Hampshire Chapter Monthly, 1st Mon., 6 p.m. Armtec Industries Shepard Dr., Grenier Field Manchester Call M. Peschke 603/774-7762 NEW MEXICO Albuquerque Chapter Monthly, 1st Thurs., 7:30 p.m. Physics & Astronomy Bldg. Univ. of New Mexico Call Rick Granfield 505/296-8651 Rochester Chapter Bi-Monthly, 4th Sat., 2 p.m. Hutchinson Hall Univ. of Rochester Call Thea Martin 716/235-0168 Rockland County Chapter Call Elizabeth Gormley Pearl River 914/735-8967 Syracuse Chapter Monthly, 3rd Wed., 7 p.m. Call Henry J. Fay 315/446-4600 OHIO Athens Chapter Call Irree Urieli 614/594-3731 Cleveland Chapter Call Gary Bergstrom 216/247-2492 Cincinnati Chapter Call Douglas Bennett 513/831-0142 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
Central Oklahoma Chapter
Monthly, 3rd Wed., 7:30 p.m.
Health Tech. Bldg., OSU Tech.
Call Larry Somers
2410 N.W. 49th
Oklahoma City, OK 73112

• OREGON
Greater Oregon Chapter
Monthly, 2nd Sat., 1 p.m.
Tektronix Industrial Park
Bldg. 30, Beaverton
Call Tom Almy
503/692-2811

• PENNSYLVANIA
Philadelphia Chapter
Monthly, 4th Sat., 10 a.m.
Drexel University, Stratton Hall
Call Melanie Hoag or Simon Edkins
215/695-7380

• TENNESSEE
East Tennessee Chapter
Monthly, 2nd Tues., 7:30 p.m.
800 Oak Ridge Turnpike, Oak Ridge
Call Richard Secrist
615/693-7380

• TEXAS
mbAustin Chapter
Contact Matt Lawrence
P.O. Box 180409
Austin, TX 78718

Dallas/Ft. Worth
Metroplex Chapter
Monthly, 4th Thurs., 7 p.m.
Call Chuck Durrett
214/245-1064

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

Perim Basin Chapter
Call Carl Bryson
Odessa – 915/337-8994

• UTAH
North Orem FIG Chapter
Contact Ron Tanner
748 N. 1340 W.
Orem, UT 84057

• VERMONT
Vermont Chapter
Monthly, 3rd Mon., 7:30 p.m.
Vergennes Union High School
Rm. 210, Monkton Rd.
Vergennes, VT
Call Don VanSyckel
802/888-6698

• VIRGINIA
First Forth of Hampton Roads
Call William Edmonds
804/898-4099

Potomac Chapter
Monthly, 2nd Tues., 7 p.m.
Lee Center
Lee Highway at Lexington St.
Arlington, VA
Call Joel Shprentz
703/860-9260

Richmond Forth Group
Monthly, 2nd Wed., 7 p.m.
154 Business School
Univ. of Richmond
Call Donald A. Full
804/739-3623

• WISCONSIN
Lake Superior FIG Chapter
Call Allen Anway
715/394-8360

MAD Apple Chapter
Contact Bill Horizon
129 S. Yellowstone
Madison, WI 53705

• FOREIGN

• AUSTRALIA
Melbourne Chapter
Monthly, 1st Fri., 8 p.m.
Contact Lance Collins
65 Martin Road
Glen Iris, Victoria 3146
03/20-2600

Sydney Chapter
Monthly, 2nd Fri., 7 p.m.
John Goodsell Bldg.
Rm. LG19
Univ. of New South Wales
Sydney
Contact Peter Tregeagle
58 Woodland Way
Morden, Surrey SM4 4DS

• CANADA
Nova Scotia Chapter
Contact Howard Harawitz
227 Ridge Valley Rd.
Halifax, Nova Scotia B3P2E5
902/477-3665

Southern Ontario Chapter
Quarterly, 1st Sat., 2 p.m.
General Sciences Bldg., Rm. 312
McMaster University
Contact Dr. N. Solntseff
Bldg. 50, Beaverton
University for Computer Science
McMaster University
Hamilton, Ontario L8S4K1
416/525-9140 ext. 3443

Toronto FIG Chapter
Contact John Clark Smith
P.O. Box 230, Station H
Toronto, ON M4C5J2

• 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
D.J. Neale
58 Woodland Way
Morden, Surry SM4 4DS

• FRANCE
French Language FIG Chapter
Contact Jean-Daniel Dodin
77 Rue du Cagire
31100 Toulouse
(16-61)44.03.06

• GERMANY
Hamburg FIG Chapter
Monthly, 4th Sat., 1500h
Contact Horst-Gunter Lysnes
Common Interface Alpha
Schwanenstrasse 27
2000 Hamburg 6

• HOLLAND
Holland Chapter
Contact: Adriaan van Roosmalen
Heusden Houtestraat 134
4817 We Breda
31 76 713104

• IRELAND
Irish Chapter
Contact Hugh Doggs
Newton School
Waterford
051/75757 or 051/74124

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

• JAPAN
Japan Chapter
Contact Toshi Inoue
Dept. of Mineral Dev. Eng.
University of Tokyo
7-3-1 Hongo, Bunkyo 113
812-2111 ext. 7073

• REPUBLIC OF CHINA
R.O.C.
Contact Ching-Tang Tzeng
P.O. Box 28
Lung-Tan, Taiwan 325

• SWITZERLAND
Swiss Chapter
Contact Max Hugelshofer
ERNI & Co., Elektro-Industrie
Stationstrasse
8206 Bruttisellen
01/833-3333

SPECIAL GROUPS

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

• FICFIGRAPH
Call Howard Pearlmutter
408/425-8700
PRESENTS

TWO NEW PUBLICATIONS
BY DR. C.H. TING

FORTH NOTEBOOK  &  INSIDE F 83

$25.00  &  $25.00

$26.00 FOREIGN SURFACE MAIL
$35.00 FOREIGN AIR MAIL

FORTH INTEREST GROUP
P. O. Box 8231
San Jose, CA  95155