FORTH Dimensions

Volume 7, Number 5

January/February 1986
$4.00

Code Modules & Data Structures

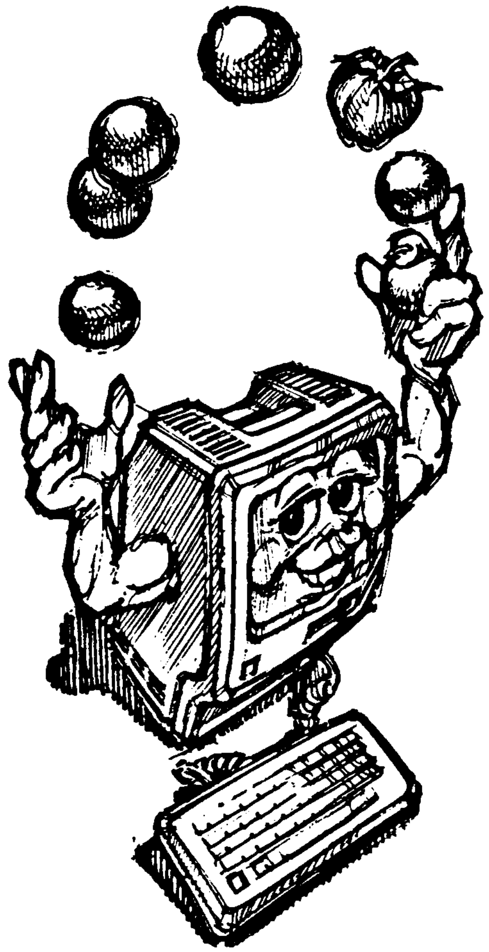
Quick DP in Forth

WALK' on Bugs

Universal Stack Word

Fast Polynomial Evaluation

The Multi-tasking Mac!



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:

Sieve:

Assembly	2-3 secs
compiled C	6-7 secs
Mach1	13 secs
other Forths	23 secs
Pascal	1270 secs

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 offers a truly interactive environment for experimentation with Forth and the Mac. The standard Motorola assembler is interactive, too, so you can also learn the assembly language for the 68000 from the keyboard. And since Mach1 uses a normal editor (on the Switcher if desired), with floating-point and local variables available, you don't have to give up the features that every other programming language has. With menu-driven templates, you can create new tasks, windows, menu bars, and even controls as easily as with a resource editor. The 200-page Forth glossary explains each Forth word; one per page with examples.

(versions for the Amiga and Atari available in early 1986)

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

FORTH Dimensions

Published by the
Forth Interest Group

Volume VII, Number 5
January/February 1986

Editor
Marlin Ouverson

Production
Cynthia Lawson Berglund

Forth Dimensions solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material published. Unless noted otherwise, material published by the Forth Interest Group (a non-profit organization) is in the public domain. Such material may be reproduced with credit given to the author and to the Forth Interest Group.

Subscription to *Forth Dimensions* is free with membership in the Forth Interest Group at $20 per year ($33 foreign air). For membership, change of address and to submit material for publication, write to: Forth Interest Group, P.O. Box 8231, San Jose, California 95155.

ISSN No. 0884-0822

Symbol Table

Simple; introductory tutorials and simple applications of Forth.

Intermediate; articles and code for more complex applications, and tutorials on generally difficult topics.

Advanced; requiring study and a thorough understanding of Forth.

Code and examples conform to Forth-83 standard.

Code and examples conform to Forth-79 standard.

Code and examples conform to fig-FORTH.

Deals with new proposals and modifications to standard Forth systems.

FORTH Dimensions

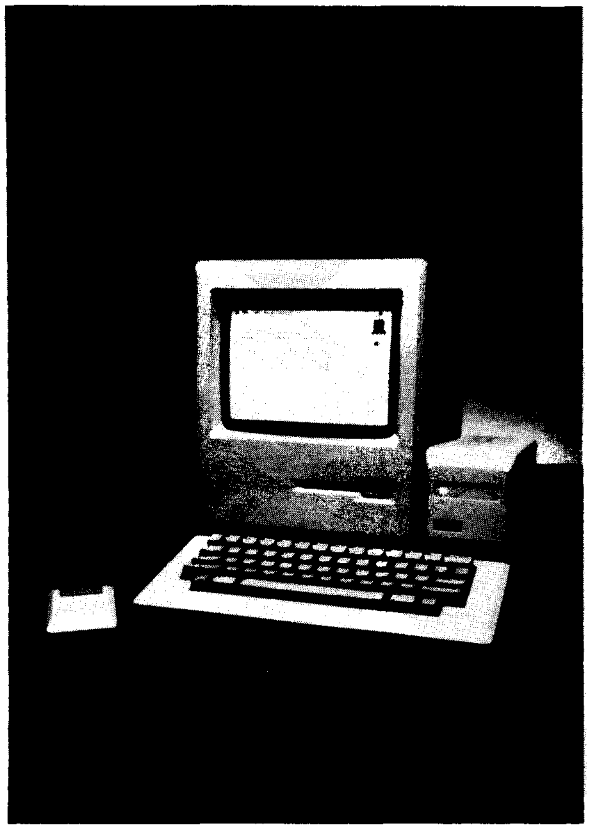
FEATURES

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 warrenty. 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 ERROR " INIT.LOCALS
   ABORT THEN  ;

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

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

```
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. Ouverson:

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 struc-

(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.

—Marlin Ouverson
Editor

(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 **!** 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 develop a word that will print numbers in a nice format: the number 12,345.67 will, for example, be printed with the comma, the decimal point, and flush-right in a display area of the width you specify.

One problem with the number-formatting words is that they need a double-precision number on the stack (as input). Many Forth novices are not interested in dealing with double-precision numbers when they are still trying to understand single-precision: how the same cell contents can be two different numbers depending on whether **U.** or . (dot) is used, but only if the number is greater than 32,767. So as part of reviewing the number-formatting words we'll also review single- and double-precision, signed and unsigned numbers.

A high-order bit of 1 in a single-precision number signals a minus sign for the signed operators (like .), but is treated as just another numeric bit by the unsigned operators (like **U.**). The interpretation depends on your choice of the operator.

Since the number-formatting words require a double-precision number, you must first convert single-precision numbers to double-precision. In doing the conversion you decide whether the number to be converted is signed or not. If the single-precision number is signed, the high-order bit — the sign bit — must be propagated across the high-order cell of the double-precision expression of that number, so that the double-precision number will have the same sign as the single-precision version. If the single-precision number is not a signed number, the high-order bit is simply a part of the number and thus is irrelevant to the bits in the (high-order) cell added to change the single-precision number to double-precision.

Most Forths have a word like **S>D** to do a signed conversion from single- to double-precision. But let's write our own to see what it's doing.

When working with bits (in this case, the high-order bit), a word that shows the bit-configuration of the datum (in this case, a number) is helpful. To see the bits, let's define the word **.BITS** as follows:

```
: .BITS  ( n - n )
  BASE @   BINARY OVER U.
  BASE ! ;
```

If your Forth does not include **BINARY** you can easily define it:

```
: BINARY   2 BASE ! ;
```

.BITS will print in base two (binary) a copy of the number on top of the stack. Since base two uses only two digits (the binary digits, abbreviated "bits"), the display will show the bit representation of the number: 0 is a bit that's off, 1 a bit that's on. As a part of normal programming hygiene, the current base is saved (on the stack) and restored at the end of the word. **.BITS** violates the common Forth rule that every word should consume its arguments (as . and **U.** do, for example). The number printed by **.BITS** remains on the stack (since what was printed was the copy made by **OVER**). But because **.BITS** will be used not within a program but interactively to look at things on the stack, and because it would be a nuisance to type **DUP .BITS** every time we wanted to look at a number, this exception seems justified.

.BITS has one drawback: it doesn't print the high-order bits if they are zero. This means that we sometimes don't see all sixteen bits in a cell. It would be nice to be able to force the printing of the high-order zero bits.

If you will tolerate a short leap forward, we can. Define **.BITS** using this word:

```
: 16BITS ( d - )
  <# # # # # # # # # # # # # # # # # #>
  TYPE ;
```

Each **#** forces the printing of a digit. Since **#** appears sixteen times, sixteen digits will be printed. This is how we force the printing of the high-order (leading) zeroes. If **16BITS** were defined as **<# #S #> TYPE** there would be no leading zeroes. The word **#>** leaves the address and count of the ASCII string version of the number, ready for **TYPE**. **16BITS** is then used to redefine **.BITS** like this:

```
: .BITS  ( n - n )
  BASE @   BINARY OVER
  0 16BITS   BASE ! ;
```

The zero converts the single-precision number to a double-precision number. The double-precision number's high-order bits (contributed by the zero we put on the stack) are all zero.

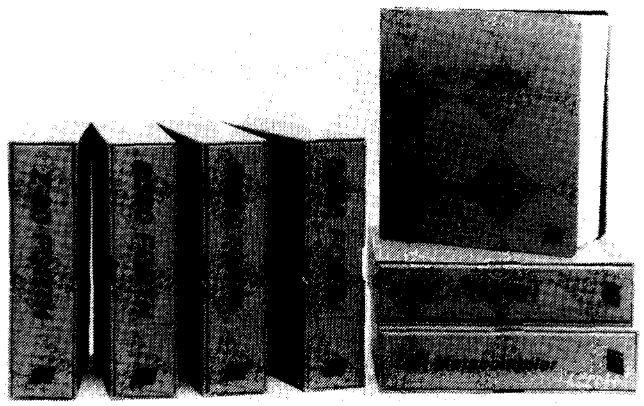
Try playing with the new **.BITS** for a while. (Things will be easier for you if you edit these examples into a disk file so that they can be easily retrieved and revised.) You can break up the string of zeroes and ones by putting the word **SPACE** into the definition of **16BITS** where you want a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

```
: 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-

TOTAL CONTROL with *LMI FORTH*™



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

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

For Applications: Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 8086, 68000, 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.

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

Overseas Distributors.
Germany: Forth-Systeme Angelika Flesch, D-7820 Titisee-Neustadt
UK: System Science Ltd., London EC1A 9JX
France: Micro-Sigma S.A.R.L., 75008 Paris
Japan: Southern Pacific Ltd., Yokohama 220
Australia: Wave-onic Associates, 6107 Wilson, W.A.

ing double-precision number, the high-order cell is zero, so the value is determined solely by the low-order cell, our original number.

To make the source code easy to read, **US>D** can be defined to convert unsigned single-precision numbers to double-precision, even though the task (and the definition) is quite simple:

```
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 number 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 **UM/MOD** 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. 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, 999999999. 2,

: #DIGITS  ( d - # )
  DABS 1 DIGITCOUNT !
  BEGIN 2DUP DIGITCOUNT @
  1- 4 * NINES + 2@ 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:

```
: #,S ( - # )
  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

BRYTE FORTH

for the

INTEL 8031

MICRO-CONTROLLER

FEATURES

—FORTH-79 Standard Sub-Set
—Access to 8031 features
—Supports FORTH and machine code interrupt handlers
—System timekeeping maintains time and date with leap year correction
—Supports ROM-based self-starting applications

COST

130 page manual	**—$ 30.00**
8K EPROM with manual	**—$100.00**

Postage paid in North America.
Inquire for license or quantity pricing.

Bryte Computers, Inc.
P.O. Box 46, Augusta, ME 04330
(207) 547-3218

ATTENTION: ENGINEERS PROGRAMMERS

PolyFORTH®II

the operating system and programming language for real-time applications involving ROBOTICS, INSTRUMENTATION, PROCESS CONTROL, GRAPHICS and more, is now available for...

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.

standard-independent by replacing **+** in this definition with **IF 1- THEN**. The **0 MAX** takes care of the situation in which the number of digits is less than the number of decimals: for example, .0023 would be stored (in integer form) as 23: **DIGITCOUNT** will be 2 and **DECIMALS** will be 4.

We can now print the number with punctuation. Let's first define the word for an unsigned double-precision number:

```
: UPRT#  ( ud - adr cnt )
   2DUP #DIGITS <#   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 (unduplicated 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 iteration 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**, forty-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 number 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.

We must also cover the other possibility 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 requires that the absolute value be used. So we get a second definition:

```
: PRT# ( d - adr cnt )
   DUP >R DABS 2DUP
   #DIGITS <# DECIMALS @ ?DUP
   IF 0 DO # LOOP ASCII . HOLD THEN
   #,S ?DUP
   IF 0 DO # # # ASCII , HOLD THEN
   #S R> SIGN #> ;
```

SIGN consumes its argument, as is typical of Forth words. We moved a copy of the top cell of the number to the return stack and then took the number's absolute value. If this is our only use of **#DIGITS** we could thus take the **DABS** out of the definition. But if you might use **#DIGITS** elsewhere, **#DIGITS** should retain its own **DABS** inside its definition.

If you wanted to force the printing of a plus for positive numbers as well as the minus for negative numbers, you could replace the word **SIGN** with **+SIGN** defined as follows:

```
: +SIGN  ( n - )
   DUP 0< IF SIGN
   ELSE IF ASCII + HOLD THEN
   THEN ;
```

To use **+SIGN** you must do some extra work in saving the "sign" part on the return stack. If the double-precision word is negative (that is, the high-order cell considered as a single-precision number is negative), then you save the high-order cell on the return stack as before. But if the high-order 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 specified width — so that the number-printing 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 PRT# 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 present, 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 character 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 (including 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 convert 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 determines the interpretation of the sign bit, we can use **d.r** for both words.

Some examples of phrases using these words:

```
TOTAL @ 12 .r
SAMPLES @ 12 U.r
CASH 2@ 25 d.r
VOLUMES 2@ 15 Ud.r
```

In each example, a number is fetched from a variable. If the number was fetched from a **2VARIABLE** 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 interpreted 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.

Michael Ham is a freelance programmer, systems designer and writer in Santa Cruz, California. This article is from a book in progress. Copyright © 1986 by Michael Ham.

1986 Rochester Forth Conference

June 10–14, 1986
University of Rochester
Rochester, New York

The sixth Rochester Forth Conference will be held at the University of Rochester, and sponsored by the Institute for Applied Forth Research, Inc. The focus will be on Real-Time Artificial Intelligence, Systems and Applications.

Call for Papers

There is a call for papers on the following topics:

- Real-Time Artificial Intelligence
- Forth Applications, including, but not limited to: real-time, business, medical, space-based, laboratory and personal systems; and Forth in silicon.
- Forth Technology, including meta-compilers, finite state machines, control structures, data structures, Forth implementations and hybrid hardware/software systems.

Papers may be presented in either platform or poster sessions. Please submit a 200 word abstract by March 31st, 1986. Papers must be received by April 30th, 1986, and are limited to a maximum of four single spaced, camera-ready pages. Longer papers may be presented at the Conference but should be submitted to the refereed *Journal of Forth Application and Research.*

Abstracts and papers should be sent to the conference chairman: Lawrence P. Forsley, Laboratory for Laser Energetics, 250 East River Road, Rochester, New York 14623. For more information, call or write Ms. Maria Gress, Institute for Applied Forth Research, 478 Thurston Road, Rochester, New York 14619 (716) 235-0168.

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

Listing 1

```
PRINT SUMMARY

Afghan, Sam          101-26-5289   8  50  15  85  92  91 150   76.6
Crash, Gonna         523-55-1967   6  76  25  75  78  76 185   81.7
Dangerous, I. M.     987-65-4321   1  30  28  92  85  92 177   80.0
Frank, Bea           887-14-9146   9  89  22  87  88  68 192   86.6
Grief, Goode         567-43-8923   5  95  29  94  86  95 180   91.9
Manners, Myna Ur     367-98-5123   6  23  12  57  65  78 165   63.4
Ong, Hang Ing        250-35-9766   3  58  23  78  82  88 184   81.4
Queliar, Kinda P.    777-99-0000   5  90  27  86  91  90 148   84.4
Turpin, Richard      425-87-5674   9 100  30 100 100 100 195   99.2
Versytl, Trighto B.  908-76-1234   8  76  28  93  88  82 182   87.1
```

Listing 2

```
PRINT MT1 SORTED
                                     MT1 ─┐
Sorting.                                  ↓
Turpin, Richard      425-87-5674   9 100  30 100 100 100 195   99.2
Grief, Goode         567-43-8923   5  95  29  94  86  95 180   91.9
Versytl, Trighto B.  908-76-1234   8  76  28  93  88  82 182   87.1
Dangerous, I. M.     987-65-4321   1  30  28  92  85  92 177   80.0
Frank, Bea           887-14-9146   9  89  22  87  88  68 192   86.6
Queliar, Kinda P.    777-99-0000   5  90  27  86  91  90 148   84.4
Afghan, Sam          101-26-5289   8  50  15  85  92  91 150   76.6
Ong, Hang Ing        250-35-9766   3  58  23  78  82  88 184   81.4
Crash, Gonna         523-55-1967   6  76  25  75  78  76 185   81.7
Manners, Myna Ur     367-98-5123   6  23  12  57  65  78 165   63.4
```

Listing 3

```
PRINT %TOTAL SORTED
                                                %TOTAL ─┐
Sorting.                                                ↓
Turpin, Richard      425-87-5674   9 100  30 100 100 100 195   99.2
Grief, Goode         567-43-8923   5  95  29  94  86  95 180   91.9
Versytl, Trighto B.  908-76-1234   8  76  28  93  88  82 182   87.1
Frank, Bea           887-14-9146   9  89  22  87  88  68 192   86.6
Queliar, Kinda P.    777-99-0000   5  90  27  86  91  90 148   84.4
Crash, Gonna         523-55-1967   6  76  25  75  78  76 185   81.7
Ong, Hang Ing        250-35-9766   3  58  23  78  82  88 184   81.4
Dangerous, I. M.     987-65-4321   1  30  28  92  85  92 177   80.0
Afghan, Sam          101-26-5289   8  50  15  85  92  91 150   76.6
Manners, Myna Ur     367-98-5123   6  23  12  57  65  78 165   63.4
```

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.

```
Scr # 39
  0 \ Application of Recursive Sort on Stack            15JAN85RHT
  1 : LOAD.STACK  ( -- n n n .. n )  \ preload stack with number
  2                                   \ for each record in file
  3    #RECORDS 0 DO I 2+ LOOP ;
  4 : BUBBLE  DEPTH 1 > IF 2DUP
  5    READ FIELD @  SWAP  READ FIELD @  \ Get scores to compare.
  6    < IF SWAP THEN      \ Swap record numbers if necessary.
  7    >R RECURSE R>  THEN ;
  8 : SORT  ( n n .. n -- n n .. n  )
  9      \ Leaves record numbers in sorted order.
 10    DEPTH 1- 0 DO BUBBLE LOOP  ;
 11 : SORTED  ( # -- )  \ Sort/display records based on field #.
 12    SET.FIELD  LOAD.STACK  SORT
 13    DEPTH 0 DO  CR READ  .SUMMARY  LOOP CR ;
 14 \ Example:  FINAL SORTED
 15
```

DASH, FIND & ASSOCIATES

Our company, DASH, FIND & ASSOCIATES, is in the business of placing FORTH Programmers in positions suited to their capabilities. We deal only with FORTH Programmers and companies using FORTH. If you would like to have your resumé included in our data base, or if you are looking for a FORTH Programmer, contact us or send your resumé to:

DASH, FIND & ASSOCIATES
808 Dalworth, Suite B
Grand Prairie TX 75050
(214) 642-5495

Committed to Excellence

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 going through the stubs and toting up results with a pocket calculator (with bad key bounce). Doggonit! I was a professional programmer, wasn't I? 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:

```
:  +.  ( 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:

```
:  +.  ( d1 n1 -- d2 )
  DUP 2 SPACES  5 .R
  S->D  D+  2DUP
  2 SPACES  6 D.R  ;
```

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:

```
:  +.  ( 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:

```
: +. ( n1 d1 n2 -- n3 d2 )
  >R ROT 1+ DUP 4 .R
  ( increment & print record count )
  ROT ROT
  ( put count under total )
  R> DUP 2 SPACES 5 .R 2 SPACES
  ( print the pledge amount )
  DUP 0= IF 1 ZERO.AMOUNTS +! THEN
  ( increment if amount is zero )
  S->D D+ 2DUP 6 D.R 2 SPACES ;
  ( calculate & print new total )
```

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 do its thing. I like the idea that the records themselves contain the directions for their own processing.

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!

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.

HS/FORTH

- Full 8087 high level support. Full range transcendentals (tan sin cos arctan logs 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
- File and/or Screen interfaces
- Segment Management
- 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 FULL MEGABYTE FORTHS!
(TEN TIMES FASTER WHEN USING AUTO-OPT!)

HS/FORTH, complete system only: $270.

Visa Mastercard

HARVARD SOFTWORKS

P.O. BOX 69
SPRINGBORO, OH 45066
(513) 748-0390

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 Metacompiler 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 Metacompiler. 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 16kb 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 buss 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.

Sample walk′ of flush; items in square brackets were typed in.

```
*****************************************
[ WALK'  FLUSH]  WALKING FLUSH
20BC  00                        LIMIT
20BE  01                  CF80  FIRST
20C0  02            CF80  BF70  -
20C2  01                  1010  B/BUF
20C4  02            1010  0400  CLIT
20C7  03      1010  0400  0004  +
20C9  02            1010  0404  /
20CB  01                  0004  1+
20CD  01                  0005  0
20CF  02            0005  0000  (DO)
20D1  00                        LIT
20D5  01                  7FFF  BUFFER
20D7  01                  C376  DROP
20D9  00                        (LOOP)
20D1  00                        LIT
20D5  01                  7FFF  BUFFER
[ DOWN]  BUFFER
2118  01                  7FFF   USE
211A  02            7FFF  205F   @
211C  02            7FFF  C778   DUP
211E  03      7FFF  C778  C778   >R
2120  02            7FFF  C778   +BUF
2122 •03      7FFF  CB7C  0808   0BRANCH
2126  02            7FFF  CB7C   USE
2128  03      7FFF  CB7C  205F   !
212A  01                  7FFF   R
212C  02            7FFF  C778   @
212E  02            7FFF  7FFF   0<
[ GO]  BOK
20D7  01                  C77A  DROP  BOK
20D9  00                        (LOOP)
20D1  00                        LIT
20D5  01                  7FFF  BUFFER
20D7  01                  CB7E  DROP
20D9  00                        (LOOP)
20D1  00                        LIT
20D5  01                  7FFF  BUFFER
20D7  01                  BF72  DROP
20D9  00                        (LOOP)
20D1  00                        LIT
20D5  01                  7FFF  BUFFER
20D7  01                  C376  DROP
20D9  00                        (LOOP)
20DD  00                        ;S
 OK
```

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 jump 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.

```
( ************************************)
(  WALK'                              )
(  BY DAVID FRANSKE                   )
(  LAST UPDATE JULY 15/84             )
( ************************************)
    FORTH DEFINITIONS DECIMAL
( ************************************)
(    KIT PEAK MODULAR PROGRAMMING     )
(             WORDS                   )
( ************************************)

: INTERNAL ( -- LATEST 10)
  LATEST 10 ;

: EXTERNAL ( -- HERE 11)
  10 ?PAIRS
  HERE 11 ;

: MODULE ( INTERNAL EXTERNAL 11 -- )
  11 ?PAIRS
  PFA LFA ! ;

( ************************************)
(      HEX FORMATERS                  )
( ************************************)

: .2H ( DATA --)
  BASE @ >R [COMPILE] HEX
  0 <# # # #> TYPE
  R> BASE ! ;

: .4H ( DATA --)
  BASE @ >R [COMPILE] HEX
  0 <# # # # # #> TYPE
  R> BASE ! ;

( ************************************)
(  BREAK UTILITY THAT CONTINUES ON    )
(       ENTERING A NUL LINE           )
( ************************************)

: BREAK ( --)
  BEGIN
    QUERY
    BL WORD HERE C@
   WHILE
    0 IN ! INTERPRET
    ."  BOK" CR
  REPEAT ;

( ************************************)
     INTERNAL
( ************************************)

  0 VARIABLE IPSAVE
  0 VARIABLE WSAVE
  0 VARIABLE SPSAVE
  0 VARIABLE NEST#
  0 VARIABLE INDENT#

(   ROUTINE SPECIFIC OUTPUT           )
```

The ForthCard™

STAND ALONE OPERATION

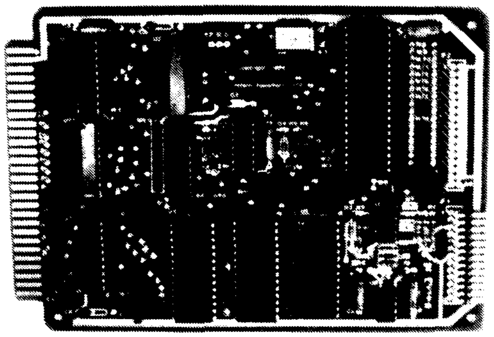
STD BUS INTERFACE

EPROM/EEPROM
PROGRAMMER

RS-232 I/O

PARALLEL I/O

ROCKWELL FORTH CHIP



Evaluation Unit **$299**
Part #STD65F11-05 includes:
ForthCard, Development
ROM, 8Kbyte RAM, Manuals

OEM Version as low as
Part #STD65F11-00 **$199**
does not include
memory or manuals

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.

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

MasterCard VISA

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 super-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 op-code 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**.

```
: .WORD ( CFA --)
  2+ NFA ID. ;

: .STACK ( --)
  DEPTH .2H SPACE
  0 3
  DO
    DEPTH I <
    IF
      5 SPACES
    ELSE
      I PICK .4H SPACE
    THEN
  -1 +LOOP  ;

: .STATUS ( --)
  IPSAVE @ 2- .4H SPACE
  .STACK
  INDENT# @ -DUP IF SPACES THEN
  WSAVE @ .WORD ;

( ***********************************)
(      VALID ONLY FOR C64-FORTH      )
(    JMP VECTORS THAT ARE CHANGED    )
( ***********************************)

: SECONDARY ;

    ASSEMBLER

NEXT 30 +                  CONSTANT NEXTJ
' ;S 7 +                   CONSTANT EXITJ
' SECONDARY CFA @ 27 + CONSTANT COLONJ
' DOES> 38 +               CONSTANT DOESJ

( ***********************************)
(      ASSEMBLY LANGUAGE SUBROUTINES  )
( ***********************************)

CREATE )LO
  0 # LDA,  1 ,X STA,
  NEXT JMP,

CREATE )HI
  1 ,X LDA, 0 ,X STA,
  ' )LO JMP,

CREATE VECTORS-OUT
  NEXT )LO # LDA,  EXITJ     STA,
                   COLONJ    STA,
  NEXT )HI # LDA,  EXITJ 1+  STA,
                   COLONJ 1+ STA,
  W 1- )LO # LDA,  NEXTJ     STA,
  W 1- )HI # LDA,  NEXTJ 1+  STA,
  PUSH )LO # LDA,  DOESJ     STA,
  PUSH )HI # LDA,  DOESJ 1+  STA,
  RTS,

( EXTENSION TO COLON )

CREATE MORECOLON
  NEST# INC,
  NEXT  JMP,

( EXTENSION TO DOES> )

CREATE MOREDOES
  NEST# INC,
  PUSH  JMP,

( EXTENSION TO ;S AND EXIT )

CREATE MOREEXIT
  NEST# DEC,
  XSAVE STX, TSX,
  SPSAVE CPX, CS
  IF,
    0 # LDA, INDENT# STA,
    VECTORS-OUT JSR, ( RESUME )
  THEN,
  XSAVE LDX,
  NEXT JMP,

( EXTENSION TO NEXT )

CREATE MORENEXT ( ADDR --)
  NEST# LDA, 0=
  IF,
    VECTORS-OUT JSR,
    IP        LDA, IPSAVE    STA,
    IP 1+     LDA, IPSAVE 1+ STA,
    W         LDA, WSAVE     STA,
    W 1+      LDA, WSAVE 1+  STA,
    ( JMP TO HIGH-LEVEL FORTH )
   HERE 11 +
   DUP )LO # LDA, IP    STA,
       )HI # LDA, IP 1+ STA,
    NEXT JMP,
    ] .STATUS CR BREAK
    [ HERE 2+ , HERE 2+ ,
    IPSAVE    LDA, IP    STA,
    IPSAVE 1+ LDA, IP 1+ STA,
    WSAVE     LDA, W     STA,
    WSAVE 1+  LDA, W 1+  STA,
    0 JSR, ( FORWARD REF VECTORS-IN )
  THEN,
  W 1- JMP,

  HERE 5 -

CREATE VECTORS-IN  HERE SWAP !
  MOREEXIT )LO  # LDA, EXITJ      STA,
  MOREEXIT )HI  # LDA, EXITJ 1+   STA,
  MORENEXT )LO  # LDA, NEXTJ      STA,
  MORENEXT )HI  # LDA, NEXTJ 1+   STA,
  MORECOLON )LO # LDA, COLONJ     STA,
  MORECOLON )HI # LDA, COLONJ 1+  STA,
  MOREDOES )LO  # LDA, DOESJ      STA,
  MOREDOES )HI  # LDA, DOESJ 1+   STA,
  RTS,

CODE WEXECUTE ( CFA --)
  XSAVE STX,
  TSX, SPSAVE STX,
  XSAVE LDX,
  VECTORS-IN JSR,
  FF #   LDA, NEST# STA,
  BOT    LDA, W    STA,
  BOT 1+ LDA, W 1+ STA,
  INX, INX, W 1- JMP,
END-CODE

: WALKABLE? ( CFA -- FLAG)
  @ LIT SECONDARY @ = ;

( ***********************************)
        EXTERNAL
( ***********************************)

: GO ( --)
  128 NEST# ! 0 SPSAVE ! ;

: DOWN ( --)
  WSAVE @ WALKABLE?
  IF
    SPSAVE  @ >R
    NEST#   @ >R
    INDENT# @ DUP >R 1+ INDENT# !
    IPSAVE  @ >R
    WSAVE  @ DUP .WORD CR WEXECUTE
    R> DUP 2+ IPSAVE ! @ WSAVE !
    R> INDENT# !
    R> NEST#   !
    R> SPSAVE  !
    .STATUS
  ELSE
    ." PRIMITIVE"
  THEN ;

: WALK ( CFA --)
  DUP WALKABLE?
  IF
    ." WALKING" DUP .WORD CR
    WEXECUTE
  ELSE
    >R CR .STACK R@ .WORD CR
    BREAK R> EXECUTE
    .STACK CR BREAK
  THEN ;

: WALK' ( -- <NAME> )
  [COMPILE] ' CFA
  STATE @
  IF
    COMPILE LIT ,
    COMPILE WALK
  ELSE
    WALK
  THEN ; IMMEDIATE

( ***********************************)
        MODULE
```

FORTH INTEREST GROUP MAIL ORDER FORM

P.O. Box 8231 San Jose, CA 95155 (408) 277-0668

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

200 - ALL ABOUT FORTH .... $25/26/35 ______
Glen B. Haydon
An annotated glossary for MVP Forth; a 79-Standard Forth.

205 - BEGINNING FORTH .... $17/18/21 ______
Paul Chirlian
Introductory text for 79-Standard.

215 - COMPLETE FORTH .... $16/17/20 ______
Alan Winfield
A comprehensive introduction including problems with answers. (Forth 79)

220 - FORTH ENCYCLOPEDIA .... $25/26/35 ______
Mitch Derick & Linda Baker
A detailed look at each FIG-Forth instruction.

225 - FORTH FUNDAMENTALS, V. 1 .... $16/17/20 ______
Kevin McCabe
A textbook approach to 79-Standard Forth.

230 - FORTH FUNDAMENTALS, V. 2 .... $13/14/16 ______
Kevin McCabe
A glossary.

232 - FORTH NOTEBOOK .... $25/26/35 ______
Dr. C. H. Ting
Good examples and applications. Great learning aid. PolyFORTH is the dialect used. Some conversion advice is included. Code is well documented.

233 - FORTH TOOLS .... $19/21/23 ______
Gary Feierbach & Paul Thomas
The standard tools required to create and debug Forth-based applications.

235 - INSIDE F 83 .... $25/26/35 ______
Dr. C. H. Ting
Invaluable for those using F-83.

237 - LEARNING FORTH.................. $17/18/21 ______
Margaret A. Armstrong
Interactive text, introduction to the basic concepts of Forth. Includes section on how to teach children Forth.

240 - MASTERING FORTH .... $18/19/22 ______
Anita Anderson & Martin Tracy (MicroMotion)
A step-by-step tutorial including each of the commands of the Forth-83 International Standard; with utilities, extensions and numerous examples.

245 - STARTING FORTH (soft cover) .... $20/21/24 ______
Leo Brodie (FORTH, Inc.)
A lively and highly readable introduction with exercises.

246 - STARTING FORTH (hard cover) .... $24/25/29 ______
Leo Brodie (FORTH, Inc.)

255 - THINKING FORTH (soft cover) .... $16/17/20 ______
Leo Brodie
The sequel to "Starting Forth". An intermediate text on style and form.

265 - THREADED INTERPRETIVE LANGUAGES $23/25/28 ______
R.G. Loeliger
Step-by-step development of a non-standard Z-80 Forth.

270 - UNDERSTANDING FORTH .... $3.50/5/6 ______
Joseph Reymann
A brief introduction to Forth and overview of its structure.

FORML CONFERENCE PROCEEDINGS

FORML PROCEEDINGS - FORML (the Forth Modification Laboratory) is an informal forum for sharing and discussing new or unproven proposals intended to benefit Forth. Proceedings are a compilation of papers and abstracts presented at the annual conference. FORML is part of the Forth Interest Group

310 - FORML PROCEEDINGS 1980 .... $30/33/40 ______
Technical papers on the Forth language and extensions.

311 - FORML PROCEEDINGS 1981 (2V) ...... $45/48/50 ______
Nucleus layer, interactive layer, extensible layer, metacompilation, system development, file systems, other languages, other operating systems, applications and abstracts without papers.

312 - FORML PROCEEDINGS 1982 .... $30/33/40 ______
Forth machine topics, implementation topics, vectored execution, system development, file systems and languages, applications.

313 - FORML PROCEEDINGS 1983 .... $30/33/40 ______
Forth in hardware, Forth implementations, future strategy, programming techniques, arithmetic & floating point, file systems, coding conventions, functional programming, applications.

314 - FORML PROCEEDINGS 1984 .... $30/33/40 ______
Expert systems in Forth, using Forth, philosophy, implementing Forth systems, new directions for Forth, interfacing Forth to operating systems, Forth systems techniques, adding local variables to Forth.

ROCHESTER PROCEEDINGS

The Institute for Applied Forth Research, Inc. is a non-profit organization which supports and promotes the application of Forth. It sponsors the annual Rochester Forth Conference.

321 - ROCHESTER 1981 (Standards Conference) $25/28/35 ______
79-Standard, implementing Forth, data structures, vocabularies, applications and working group reports.

322 - ROCHESTER 1982
(Data bases & Process Control) .... $25/28/35 ______
Machine independence, project management, data structures, mathematics and working group reports.

323 - ROCHESTER 1983 (Forth Applications) . $25/28/35 ______
Forth in robotics, graphics, high-speed data acquisition, real-time problems, file management, Forth-like languages, new techniques for implementing Forth and working group reports.

324 - ROCHESTER 1984 (Forth Applications) . $25/28/35 ______
Forth in image analysis, operating systems, Forth chips, functional programming, real-time applications, cross-compilation, multi-tasking, new techniques and working group reports.

325 - ROCHESTER 1985
(Software Management and Engineering) $20/21/24 ______
Improving software productivity, using Forth in a space shuttle experiment, automation of an airport, development of MAGIC/L, and a Forth-based business applications language, includes working group reports.

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 _____

602 - POSTER (BYTE Cover) .... $15/16/18 _____

616 - HANDY REFERENCE CARD .... FREE _____

683 - FORTH-83 HANDY REFERENCE CARD ...... FREE _____

FORTH INTEREST GROUP WINTER SPECIAL

Dr. Dobb's Journal Annual Forth Issues
3 for $10.00 9/82, 9/83, 9/84
USA-Canada only $10/12/14

3 FOR
$10.00
(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 _____________________________________

____

Phone _______________________________________

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) _______________

____

FORTH INTEREST GROUP

P.O. BOX 8231 SAN JOSE, CALIFORNIA 95155 408/277-0668

Name ______________________

Company ______________________

Address ______________________

City ______________________

State/Prov. __________ ZIP __________

Country ______________________

Phone ______________________

OFFICE USE ONLY

By ________ Date ________ Type ________

Shipped By ________ Date ________

UPS Wt. ________ Amt. ________

USPS Wt. ________ Amt. ________

BO Date ________ Wt. ________ Amt. ________

By ________

ITEM #	TITLE	AUTHOR	QTY	UNIT PRICE	TOTAL
107	MEMBERSHIP				SEE BELOW

☐ Check enclosed (payable to: **FORTH INTEREST GROUP**)

☐ VISA ☐ MASTERCARD

Card # ______________________

Expiration Date ______________________

Signature ______________________

SUBTOTAL	
10% MEMBER DISCOUNT MEMBER # __________	
CA. RESIDENTS SALES TAX	
HANDLING FEE	**$2.00**
MEMBERSHIP FEE $20/27/33 ☐ NEW ☐ RENEWAL	
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 6%. San Francisco Bay Area add 7%.

1/86

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

BVARIABLE is a defining word which, when executed, takes two arguments from the stack: the address of a pointer and an offset value. **BVARIABLE** 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.

BVARIABLE 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.

BVARIABLE 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 **BVARIABLE** is defined in the library support system, so it is always available. A programmer who writes a library module uses **BVARIABLE** 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 BVARIABLE Works

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

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

```
: BVARIABLE \name    ; aptr noffset --
            \;P Creates a based variable
            \Child:  -- address
 CREATE , ,
  ;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
```

When this word is executed, it creates the new word in the dictionary (with **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 **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 **DOES>**. **DUP** @ puts the offset value on the stack. **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 **;CODE** version of **BVARIABLE**, only for the 8086/8088 and only for F83 (not necessarily for other Forth-83 systems for the 8086/8088), is:

```
: BVARIABLE \name    ; aptr noffset --
            \;P Creates a based
            \variable
            \Child:  -- address
 CREATE , ,
  ;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
```

A full explanation of this definition would divert us into the internals of this particular Forth system. But we might mention that **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 **W** as a scratch register which does not need to be saved and restored.

Here is a sample session which illustrates the use of **BVARIABLE** (either version, as they

```
VARIABLE PTR
PTR 0 BVARIABLE X
PTR 15 BVARIABLE Y
20000 PTR !
X . 20000
Y . 20015
25000 PTR !
X . 25000
Y . 25015
: T    30000 0 DO    X DROP    LOOP ;
```

The crude benchmark in the last line shows the speed improvement with the **;CODE** version. An ordinary variable, a **BVARIABLE** with the **;CODE** definition, and a **BVARIABLE** with the **CREATE ... DOES>** definition, were used as the **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 **DROP** and to loop back. Clearly, the **;CODE** version would be recommended for practical use.

Write it once!

MasterFORTH

Portable programming environment

Whether you program on the **Macintosh,** the **IBM PC**, an **Apple II** series, a **CP/M** system, or the **Commodore 64**, your program will run unchanged on all the rest. If you write for yourself, MasterFORTH will protect your investment. If you write for others, it will expand your marketplace.

MasterFORTH is a state-of-the art implementation of the Forth computer language. Forth is interactive – you have immediate feedback as you program, every step of the way. Forth is fast, too, and you can use its built-in macro assembler to make it even faster. MasterFORTH's relocatable utilities, transient definitions, and headerless code let you pack a lot more program into your memory. The resident debugger lets you decompile, breakpoint, and trace your way through most programming problems. A string package, file interface, and full screen editor are all standard features. And the optional target compiler lets you optimize your application for virtually any programming environment.

MasterFORTH exactly matches the Forth-83 Standard dialect described in *Mastering Forth* by Anderson and Tracy (Brady, 1984). The standard package includes the book and over 100 pages of supplementary documentation.

MasterFORTH standard package	
Macintosh	$125
IBM PC & PC Jr. (MS DOS 2.1)	125
Apple II, II+, IIe, IIc (DOS 3.3)	125
CP/M 2.x (IBM 3740 8")	125
Commodore 64 (with graphics)	100
Extensions	
Software Floating Point	$60
Hardware (8087) Floating Point	60
Graphics (selected systems)	60
Relocator (with utility sources)	60
Target compiler (RAM and ROM)	350
Target Application Generation System (TAGS) - MasterFORTH, Target compiler & relocator	495
Publications	
Printed source listings (each)	$15
Forth-83 International Standard	15

NEW PRODUCTS

MODEL List Processor.... $40
(with ELIZA and micro-LISP)

MICROMOTION
8726 S. Sepulveda Bl., #A171
Los Angeles, CA 90045
(213) 821-4340

A Universal Stack Word

Doneil 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 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:

: DROP	**STACK A|** ;
: DUP	**STACK A|AA** ;
: SWAP	**STACK AB|BA** ;
: ROT	**STACK ABC|BCA** ;
: 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.

```
Screen # 1
( run-time stack words                                  05/06/85 )

: (STACK) ( Run-time STACK in forth-83 )
            R@ C@ O
   DO       I 2*  HERE + !  LOOP   R@ 1+ DUP C@ + 1+ R@ 2+
   ?DO      I C@  HERE + @  LOOP   R> 1+ COUNT  + >R  ;

CODE (STACK) ( Run-time STACK in 8086 assembler )
           CL, [SI]    MOV        CH, CH   XOR
           DI, DP      MOV        BX, DI   MOV
   1$:     AX          POP        WORD     STOS      1$  LOOP
           SI          INC        CL, [SI] MOV
           AH, AH      XOR        SI       INC       3$  JCXZ
   2$:     BYTE        LODS       DI, AX   MOV
           DX, [BX+DI] MOV        DX       PUSH      2$  LOOP
   3$:     NEXT


Screen # 2
( char>   stack                                         05/06/85 )

: CHAR> ( -- <n> {Get next ascii character from input stream} )
      >IN @ 1 >IN +! BLK @ ?DUP IF BLOCK ELSE TIB THEN + C@ ;

: STACK ( -- abcd|abcd {Perform stack rearrangement} )
           ?COMP COMPILE (STACK)           \ compile run-time word
   BEGIN  CHAR> BL <> UNTIL                \ find stack picture
       O                                   \ counter for #input
   BEGIN  1+ CHAR> ASCII | = UNTIL         \ compile #input items
       DUP >R C,  >IN @   -1               \ remember where we are
   BEGIN  1+ CHAR> BL = UNTIL  C,          \ compile #output items
       >IN !                               \ back to output items
   BEGIN CHAR> DUP BL <>     WHILE         \ while valid output
       64 - R@ SWAP - 2* C,  REPEAT        \ compile them
       R> 2DROP ; IMMEDIATE                \ clr stacks


Screen # 3
( box1 box2   star1  star2                              05/06/85 )

: BOX1 ( x1 y1 x2 y2 -- draw box ) ( 43 bytes | 20.9 seconds )
        2OVER 3 PICK OVER 2DUP 2ROT LINE 2OVER LINE
        3 PICK OVER 2DUP 2ROT LINE LINE ;

: BOX2 ( x1 y1 x2 y2 -- draw box ) ( 35 bytes | 20.1 seconds )
        STACK ABCD|ABCBCBCDADCDABAD   4 LINES ;

: STAR1 ( pt1...pt5 -- draw star ) ( 88 bytes | 43.6 seconds )
      9 PICK 9 PICK 7 PICK 7 PICK 2DUP 2ROT LINE 2OVER LINE
      7 PICK 7 PICK 2DUP 2ROT LINE 2OVER LINE 7 PICK 7 PICK
      LINE 2DROP 2DROP 2DROP ;

: STAR2 ( pt1...pt5 -- draw star ) ( 42 bytes | 41.8 seconds )
      STACK ABCDEFGHIJ|ABEFEFIJIJCDCDGHGHAB   5 LINES ;
```

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 questionaire below and return it directly to us by March 15! All completed questionaires 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 ______301-400 ______401-600 ______?

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[7] 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 **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:

```
: 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 polynomial does have special features, there may very well be an algorithm 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 nth 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.

Horner's Scheme

Horner's scheme is contained in the algebraic identity

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots a_1 x + a_0 = (\ldots ((a_n x + a_{n-1})x + a_{n-2})x \ldots + a_1)x + a_0$$

The expression on the right side is evaluated from the inside out. The calculation can be set out in algorithmic form[2, 9]. Let $b_n = a_n$ and compute $b_{n-1}, b_{n-2}, \ldots, b_1, b_0$ by the recurrence $b_k = a_k + b_{k+1} * x_0$. 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 $p'(x \quad 10) = q(x_0)$. 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 $y_n = a_n$ and $z_n = a_n$. Compute successively the quantities

$y_k = x_0 * y_{k+1} + a_k$ and $z_k = x_0 * z_{k+1} + y_k$ 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 **HORNER_SCHEME** and **FHORNER_SCHEME** to realize the Horner polynomial evaluation schemes in integer and floating-point arithmetic, respectively, and a word **NEWTON_CYCLE** including evaluation of a polynomial and its derivative simultaneously in floating-point arithmetic.

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

$$x_{n+1} = x_n - f(x_n)/f'(x_n)$$

```
: DEFER
\ create a deferred definition
   CREATE   ( --- )
     2 ALLOT
   DOES>    ( ? )
     @ EXECUTE   ;

: (IS)   ( value --- )
\ the run-time action of IS
   R@ @  >BODY !   ( retrieve target address )
   R> 2+ >R   ;

: IS   ( value --- )
\ sets the body of the following word to value.
\ If compiling, the word set is the one which follows in the
\ definition.
   STATE @   ( compiling? )
   IF    COMPILE (IS)   ( run-time action )
   ELSE  ' >BODY !      ( interpretive action )
   THEN   ;  IMMEDIATE
```

Figure One

```
SCR # 1
 0 \ Loader                                        FORTH-83  03DEC84NG
 1
 2 : HORNER-MARKER   ( null )   ;
 3
 4
 5  2 11 THRU
 6
 7
```

If certain technical conditions[2] 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[6]. The implementation presented here will be in floating-point

```
SCR # 2
\ Poly-array                                FORTH-83  02DEC84NG

: POLY_ARRAY
  CREATE  ,   ( n --- )
\ The degree n will be followed by the n+1 integer coefficients
  DOES>       ( n --- nth coeff if n >= 0, degree if n < 0 )
   SWAP DUP 0<
    IF     DROP @          ( degree )
    ELSE   2* 2+ + @       ( nth coefficient )
    THEN   ;

DEFER COEFFICIENT


```

```
SCR # 3
\ Horner_scheme                             FORTH-83  02DEC84NG

: HORNER_SCHEME   ( n --- n1 = p[n] )
  -1 COEFFICIENT  ( degree)  DUP COEFFICIENT  ( leading coeff)
  SWAP 1-  0 SWAP DO
   OVER * I COEFFICIENT +
  -1 +LOOP   ;


```

```
SCR # 4
\ Examples                                  FORTH-83  02DEC84NG

3 POLY_ARRAY PQ
 -6 , 11 , -6 , 1 ,

: PQ()   ( n --- n1 = pq[n] )
  ['] PQ IS COEFFICIENT
  HORNER_SCHEME   ;

6 POLY_ARRAY RS
 1 , 1 , 1 , 1 , 1 , 1 , 1 ,

: RS()   ( n --- n1 = rs[n] )
  ['] RS IS COEFFICIENT
  HORNER_SCHEME   ;

```

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

FORTH

FREEDOM OF CHOICE

79 STANDARD

fig MODEL

IBM®

TRS-80

CP/M

SOTA Computing Systems Limited lets you choose between either the versatile figFORTH model or the popular 79 Standard. Each version is available for a number of popular computer systems including the IBM PC, XT and AT (or compatibles), the TRS-80 Model I, III and 4/4P, or any computer system running CP/M (version 2.x) or CP/M Plus (version 3.x). What's more, SOTA doesn't require you to enter into any awkward or expensive royalty or licensing arrangements. As long as your applications programs do not offer the end user access to the basic FORTH system, you are free to make as many copies of the compiled FORTH system as you please and distribute them as you wish. FORTH from SOTA is the FORTH of choice for both the novice and experienced programmer. Make it your choice now! Order your copy today.

When you order from SOTA, both the fig model and 79 standard come complete with the following extra features at no additional charge:

• full featured string handling • assembler • screen editor • floating point • double word extension set • relocating loader • beginner's tutorial • comprehensive programmer's guide • exhaustive reference manual • unparalleled • technical support • source listings • • unbeatable price •

ORDER FORM

GENTLEMEN: Rush me my order! **TOTAL** US funds

☐ Enclosed is my: ☐ check ☐ money-order

☐ Please bill my: ☐ VISA ☐ MasterCard

for $89.95

Please send me: ☐ 79 Standard FORTH ☐ figFORTH model for the:

☐ IBM PC ☐ XT ☐ AT (and compatibles)

☐ TRS-80 Model I ☐ Model III ☐ Model 4 ☐ Model 4P

☐ CP/M Version 2.x ☐ CP/M Plus (Version 3.x)

For CP/M versions please note: 5 1/4" formats only and please specify computer type:

NAME:

STREET:

CITY/TOWN:

STATE: ZIP:

CARD TYPE: EXPIRY:

CARD NO:

SIGNATURE:

ORDER TODAY 213-1080 Broughton Street Vancouver, British Columbia Canada • V6G 2A8

MasterCard • Order by Mail or Phone (604) 688-5009 • VISA

State of the Art since 1981

SOTA Computing Systems Limited

IBM, TRS-80 and CP/M are registered trademarks of International Business Machine Corporation, Radio Shack and Digital Research respectively

```
SCR # 5
\ Poly_farray                                    FORTH-83  02DEC84NG

: F,   ( r --- ) \ stores r in first available dictionary slot
   HERE  F#BYTES ALLOT  F!   ;

: POLY_FARRAY
   CREATE  ,    ( n --- )
\ the degree n will be followed by the n+1 real coefficients
   DOES>        ( n --- nth coeff if n >= 0, degree if n < 0 )
    SWAP DUP 0<
     IF    DROP @
     ELSE  F#BYTES * 2+ + F@
     THEN   ;

DEFER FCOEFFICIENT



SCR # 6
\ Floating Horner scheme                         FORTH-83  02DEC84NG

: FHORNER_SCHEME   ( r --- r1 = p[r] )
        -1 FCOEFFICIENT   ( degree )
    DUP >R FCOEFFICIENT   ( leading coefficient )
        R> 1-  0 SWAP     ( r leading_coeff 0 deg-1 )
    DO
     FOVER F*  I FCOEFFICIENT  F+
    -1 +LOOP
    FSWAP FDROP   ;
```

arithmetic. 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[4]. Screen 0 is merely descriptive and cannot be **LOAD**ed. 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*[3], 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_ARRAY** prefaces 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.

```
SCR # 7
\ Examples                                   FORTH-83  02DEC84NG

3 POLY_FARRAY FPQ
 -6E F,  11E F,  -6E F,  1E F,

: FPQ()   ( r --- r1 = fpq[r] )
   ['] FPQ IS FCOEFFICIENT
   FHORNER_SCHEME   ;

4 POLY_FARRAY EXP4   ( exponential series through 4th power )
 1E F,  1E F,  .5E F,  1.66666667E-1 F,  4.16666667E-2 F,

: EXP4()  ( r --- r1 = exp4[r] )
   ['] EXP4 IS FCOEFFICIENT
   FHORNER_SCHEME   ;


SCR # 8
\ Newton's method for roots of polynomials  FORTH-83  03DEC84NG

: NEWTON_CYCLE   ( x --- x1 = x - p[x]/p'[x])
      -1 FCOEFFICIENT
  DUP >R FCOEFFICIENT FDUP                          ( x an an )
  R> 1- 1 SWAP
  DO                                         ( from n-1 down to 1 )
   FROT FROT FOVER F*  I FCOEFFICIENT  F+          ( zI+1 x yI )
   FROT FROT FSWAP FOVER F* FROT FDUP FROT F+      ( x yI zI )
  -1 +LOOP                                         ( x yI zI )
  FROT FROT FOVER F*  0 FCOEFFICIENT  F+ FROT  ( x p[x] p'[x] )
  F/ F-   ;                                    ( x - p[x]/p'[x] )


```

Screen 2 Here is the first of the new data types. As illustrated on screen 4, the defining word is used in the form

<degree> **POLY_ARRAY** <name>

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

| deg | a_0 | a_1 | . . . | a_{n-1} | a_n |

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

Screen 3 The deferred word **COEFFICIENT** in screen 2 will be the target for the vectored arrays of coefficients. Its immediate use is to allow writing the word **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 **HORNER_SCHEME**. The first is the polynomial

$pq(x) = x^3 - 6x^2 + 11x - 6 = (x - 1)(x - 2)(x - 3)$

SOFTWARE for the HARDCORE

MasterFORTH

FORTH-83 STANDARD

• • 6809 Systems available for
FLEX disk sustems $150
OS9/6809 . . . . $150

• • 680x0 Systems available for
MACINTOSH $125
CP/M-68K $150

• • tFORTH/20 for 68020 Single Board Computer

Disk based development system under OS9/68K . . . $290
EpROM set for complete stand-alone SBC $390

• • Forth Model Library - List handler, spreadsheet, Automatic structure charts . . . each . $40

• Target compilers : 6809, 6801, 6303, 680x0, 8088, Z80, 6502

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

HARDWARE for the HARDCORE

68020 SBC, 5 1/4" floppy size board with 2MB RAM, 4 x 64K EpROM sockets, 4 RS232 ports, Centronics parallel port, timer, battery backed date/time, interface to 2 5 1/4" floppies and a SASI interface to 2 winchester disks $2750
68881 flt pt option $500
OS9 multitask&user OS . . $350

FAST! int. benchmarks speeds are
2 x a VAX780, 10 x an IBM PC

UBZ FORTH™

for the Amiga™

- *FORTH-83 compatible
- *32 bit stack
- *Multi-tasking
- *Separate headers
- *Full screen editor
- *Assembler
- *Amiga DOS support
- *Intuition support
- *ROM kernel support
- *Graphics and sound support
- *Complete documentation
- *Assembler source code included
- *Monthly newsletter

$85

Shipping included
in continental U.S.
(Ga. residents add sales tax)

UBZ *Software*

(404)-948-4654

(call anytime)
or send check or money order to:

UBZ *Software*
395 St. Albans Court
Mableton, Ga. 30059

*Amiga is a trademark for Commodore Computer. UBZ FORTH is a trademark for UBZ Software.

```
SCR # 9
 \ Newton's scheme for roots of polynomials   FORTH-83  03DEC84NG

 FVARIABLE TOLERANCE    VARIABLE #CYCLES

 : NEWTON'S_SCHEME   ( starting guess --- root; both floating )
    1E-7 TOLERANCE F!  10 #CYCLES !
    BEGIN             ( the Newton iterations )
     -1 #CYCLES +!
     FDUP NEWTON_CYCLE        ( iterate )
     FSWAP FOVER  F- FABS
     TOLERANCE F@ F<  #CYCLES @ 0=  OR
    UNTIL   ;                 ( close enough? cycled out? )



SCR # 10
 \ Newton's method -- examples                FORTH-83  03DEC84NG

 2 POLY_FARRAY X*X-2
  -2E F,  0E F,  1E F,

 : FIND_SQRT(2)
    ['] X*X-2 IS FCOEFFICIENT
    NEWTON'S_SCHEME   ;

 : SOLVE_FPQ
    ['] FPQ IS FCOEFFICIENT
    NEWTON'S_SCHEME   ;


```

The second is

$rs(x) = 1 + x + x^2 + \ldots + x^6 = (x^7 - 1)/(x - 1)$

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[4]. The word **F,** is analogous to the integer word , and is not included in the proposed standard. The words **POLY_FARRAY** and **FCOEFFICIENT** are completely analogous to the earlier words **POLY_ARRAY** and **COEFFICIENT**.

Screen 6 Because the stack contains numbers of mixed types, generalization of **HORNER_SCHEME** to the analogous **FHORNER_SCHEME** has a little hitch: a simple **SWAP** on a stack holding sixteen-bit integers requires a temporary storage on the return stack when the data stack holds numbers of mixed types. (This avoids appeal to the particular implementation of the floating-point numbers.)

```
SCR # 11
 0 \ Newton's method -- more examples          FORTH-83  03DEC84NG
 1
 2 10 POLY_FARRAY COS10
 3 \ poly approximation to cosine, error < 2E-9 for 0 <= x <= pi/2
 4  1E F,  0E F,  -.50000000E F,  0E F,  .0416666642E F,  0E F,
 5  -.0013888397E F,  0E F,  .0000247609E F,  0E F, -2.605E-7 F,
 6
 7 : SOLVE_COS10
 8    ['] COS10 IS FCOEFFICIENT
 9    NEWTON'S_SCHEME   ;
10
11
12
13
14
15
```

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, necessitating many iterations, the process is set to stop after a certain **#CYCLES**. The values of **TOLERANCE** and **#CYCLES** should be inserted into the word **NEWTON'S_SCHEME** before it is used. Choose a starting value x_0, a floating-point number, and execute **NEWTON'S_SCHEME**. If the pitfalls mentioned earlier have been avoided, the root will soon appear on the stack, from which it can be pulled by **F.**

Screen 10 The square root of two is the positive root of the polynomial $x^2 - 2$. The words **FIND_SQRT(2) F.** will display $2^{1/2}$ when applied to initial values such as 1E and 2E. When used on -1E, they print out $-2^{1/2}$. The polynomial pq (or fpq, its floating-point incarnation) has the obvious roots 1, 2 and 3, so serves as a good testing ground. Enter various initial guesses and use **SOLVE_FPQ** to see which root is the limit of the iterates.

Screen 11 This example shows a polynomial optimized to represent the cosine function on a fixed interval. The cosine function is often defined through its power series, of which the terms up to and including the tenth power are

$$\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - x^{10}/10!$$

If $0 =< x =< pi/2$, the error in the optimized polynomial **COS10** is no bigger in absolute value than 2E-9. Many such optimized polynomials can be found[1], from which I took this one. The smallest positive root of the equation cos x = 0 is the number pi/2. It might be guessed that the smallest positive root of the approximate polynomial should approximate pi/2. Use **1.5E SOLVE_COS10 F.** to test the strength of this conjecture. (Notice that the polynomial cos10 is not generic: it is in fact a polynomial of degree five in the variable x^2 and therefore it has an evaluation in fewer multiplications than the generic algorithm.)

Steven Ruzinsky described a systematic way to generate optimized polynomial approximations and gave a Forth implementation of his method in a recent article[8].

Example

To show how these words can be used interactively from the keyboard, 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 1664[5]. 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_FARRAY VIETA
-14356197E F,  30E F,  0E F,  1E F,
' VIETA IS FCOEFFICIENT
```

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

```
200E NEWTON'S_SCHEME 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

1. Abramowitz, Milton and Irene A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards Applied Mathematics Series 55, Washington, 1955.
2. Burden, Richard L., J. Douglas Faires and Albert C. Reynolds, *Numerical Analysis*, 2nd ed., Prindle, Weber and Schmidt, Boston, 1981.
3. Anderson, Anita, and Martin Tracy, *Mastering Forth*, Brady, Bowie, MD, 1984.
4. Duncan, Ray and Martin Tracy, "The FVG Standard Floating-Point Extension," *Dr. Dobb's Journal*, September 1984.
5. Goldstine, Herman H., *A History of Numerical Analysis from the 16th Century Through the 19th Century*, Springer-Verlag, New York, 1977.
6. Grossman, Nathaniel, "Newton's Method: Fixed-Point Square Roots in Forth," *Forth Dimensions*, March/April 1984.
7. La Quey, Robert, "Reverse Polish Translation," *1984 FORML Conference Proceedings*, Forth Interest Group, San Jose, CA, 1984.
8. Ruzinsky, Steven A., "A Simple Minimax Algorithm," *Dr. Dobb's Journal*, July 1984.
9. Sedgewick, Robert, *Algorithms*, Addison-Wesley, Reading, MA, 1983.

SOFTWARE COMPOSERS

IT'S HERE!

The DELTA BOARD

"I'm delighted to see Software Composers' board on the market. It provides incredible capability and versatility with minimal parts, size, and price. An excellent introduction to the new generation of hardware and software."

Chuck Moore
November, 1985

=============== SC-1000C DELTA BOARD ==============
The SC-1000C Delta Board is the first low cost cpu board available using the super-fast Novix chip. The Delta Board comes fully assembled, tested, and ready to use with a 30 day warranty. The Delta Board comes with a manual, a 9600 baud serial port on board, a terminal connector and cable, a 5 volt power supply, and a mounting board. Orders filled on a first come, first served basis.

Ready to use--turn it on and start development!!

============= DELTA BOARD INFORMATION =============
* **NOVIX NC4000P chip on board. (4 Mhz operation).**
* **cmFORTH interpreter and compiler in EPROM.**
* **4K 16 bit words of static RAM, 4K words of EPROM.**
* **21 independently controllable I/O ports on bus.**
* **Up to eight 256 word data stacks and 256 word return stacks can be selected for multi-tasking.**
* **Delta Board measures 4 1/2" x 6 1/2" with a 72 pin edge connector bus with all major Novix signals.**

================ PRICE INFORMATION ================
```
 SC-1000C DELTA BOARD:  $895
  FIG MEMBER DISCOUNT:   -45
                       ------
                        $850
  LOT ORDER DISCOUNT:    -75 (ORDERED IN LOTS OF
                       ------     FOUR OR MORE)
                        $775
```

If you would like additional information on how to order, send your name and address to:

SOFTWARE COMPOSERS
210 California St., Suite F,
Palo Alto, CA 94306

7th Annual Conference

FORML at Asilomar

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 concensus 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 decompilers, 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 concensus 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/algorithms to which students can refer when working on their own projects. An opportunity for Forth instruction was perceived at community colleges and at elementary and high schools, which often welcome outside contributions to the curriculum and to students' extended studies.

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 both 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 presentors 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

FOR TRS-80 MODELS 1, 3, 4, 4P IBM PC/XT, AT&T 6300, ETC.

WHICH FORTH has all the POWERFUL APPLICATIONS?

- **DATAHANDLER** database
- **FORTHWRITE** word processor
- **FORTHCOM** communications
- **GENERAL LEDGER** accounting
- **GAMES** for fun and technique
- **EXPERT-2** expert system
- **TRADESHOW** commodities terminal
- **GRAPHICS, 8087** support, many other utilities

You've Been Thinking About It. Isn't It Time to Put It to Work?

MMSFORTH

The total software environment for IBM PC/XT, TRS-80 Model 1, 3, 4 and close friends.

- Personal License (required):

MMSFORTH V2.4 System Disk	**$179.95**

(TRS-80 Model 1 requires lowercase, DDEN, 1 40-track drive.)

- Personal License (additional modules):

FORTHCOM communications module	**$ 49.95**
UTILITIES	**49.95**
GAMES	**39.95**
EXPERT-2 expert system	**69.95**
DATAHANDLER	**59.95**
DATAHANDLER-PLUS (PC only, 128K req.)	**99.95**
FORTHWRITE word processor	**99.95**

- Corporate Site License Extensions . . . . **from $1,000**
- Bulk Distribution . . . **from $500/50 units.**
- Some recommended Forth books:

STARTING FORTH (programming)	**19.95**
THINKING FORTH (technique)	**15.95**
BEGINNING FORTH (re MMSFORTH)	**16.95**

Shipping/handling & tax extra. No returns on software.

Ask your dealer to show you the world of MMSFORTH, or request our free brochure.

MILLER MICROCOMPUTER SERVICES
61 Lake Shore Road, Natick, MA 01760
(617) 653-6136

FIG Chapters

U.S.

• ALABAMA

Huntsville FIG Chapter
Call Tom Konantz
205/881-6483

• ALASKA

mbKodiak Area Chapter
Call Horace Simmons
907/486-5049

• ARIZONA

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

Tucson Chapter
Twice Monthly,
2nd & 4th Sun., 2 p.m.
Flexible Hybrid Systems
2030 E. Broadway #206
Call John C. Mead
602/323-9763

• ARKANSAS

Central Arkansas Chapter
Twice Monthly, 2nd Sat., 2p.m. &
4th Wed., 7 p.m.
Call Gary Smith
501/227-7817

• CALIFORNIA

Los Angeles Chapter
Monthly, 4th Sat., 10 a.m.
Hawthorne Public Library
12700 S. Grevillea Ave.
Call Phillip Wasson
213/649-1428

Monterey/Salinas Chapter
Call Bud Devins
408/633-3253

Orange County Chapter
Monthly, 4th Wed., 7 p.m.
Fullerton Savings
Talbert & Brookhurst

Fountain Valley
Monthly, 1st Wed., 7 p.m.
Mercury Savings
Beach Blvd. & Eddington
Huntington Beach
Call Noshir Jesung
714/842-3032

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

Sacramento Chapter
Monthly, 4th Wed., 7 p.m.
1798-59th 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 Cottilion 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
312/879-3242

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 Flones
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

• NEW YORK

FIG, New York
Monthly, 2nd Wed., 8 p.m.
Queens College
Call Ron Martinez
212/517-9429

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 Isreal Urieli
614/594-3731

Cleveland Chapter
Call Gary Bergstrom
216/247-2492

Cincinatti 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. 50, 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
215/895-2628

• TENNESSEE

East Tennessee Chapter
Monthly, 2nd Tue., 7:30 p.m.
Sci. Appl. Int'l. Corp., 8th Fl.
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

Periman 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
802/388-6698

• 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 Horzon
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
03/29-2600

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

• BELGIUM

Belgium Chapter
Monthly, 4th Wed., 20:00h
Contact Luk Van Loock
Lariksdreff 20
2120 Schoten
03/658-6343

Southern Belgium FIG Chapter
Contact Jean-Marc Bertinchamps
Rue N. Monnom, 2
B-6290 Nalinnes
Belgium
071/213858

• 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
Unit 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.,
7p.m., Rm. 408
Polytechnic of South Bank
Borough Rd., London
D.J. Neale
58 Woodland Way
Morden, Surry SM4 4DS

• FRANCE

French Language 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 Lynsche
Common Interface Alpha
Schanzenstrasse 27
2000 Hamburg 6

• HOLLAND

Holland Chapter
Contact: Adriaan van Roosmalen
Heusden Houtsestraat 134
4817 We Breda
31 76 713104

FIG des Alpes Chapter
Contact: Georges Seibel
19 Rue des Hirondelles
74000Annely
50 57 0280

• 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
Stationsstrasse
8306 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

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

FIGGRAPH
Call Howard Pearlmutter
408/425-8700

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

Address Correction Requested