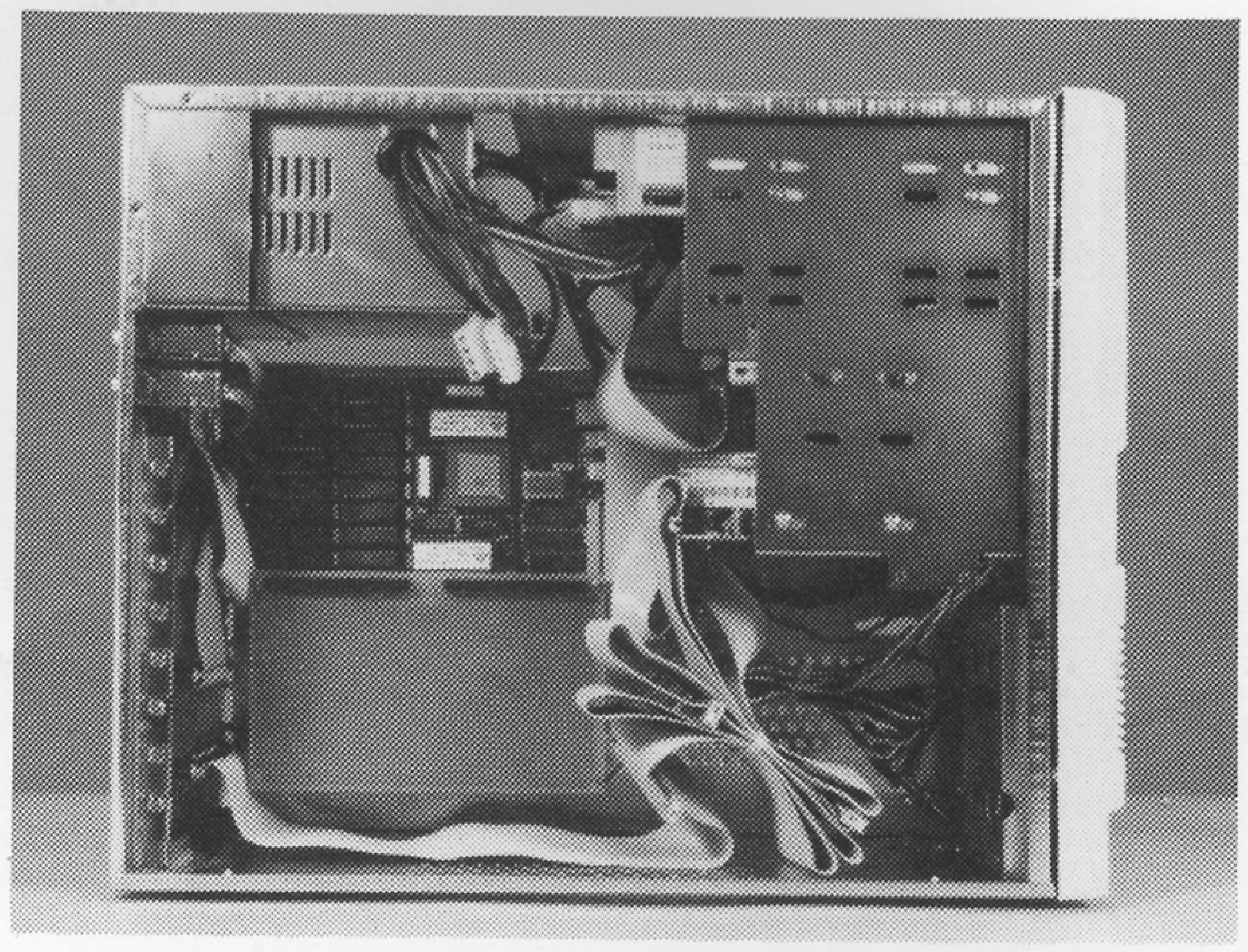
VOLUME XII, NUMBER 1 MAY/JUNE 1990

FORTH DIMENSIONS

SILICON COMPOSERS

SC/FOX™ BOX Computer System



The SC/FOX (Forth Optimized eXpress) BOX is a high-speed stand-alone computer, designed both as a real-time development platform and a system controller for embedded applications. Mounted inside is the SC/FOX Single Board Computer, a small (100x160mm) stand-alone computer that uses the 16-bit RTX 2000 Forth CPU, and the SCSI I/O board. Communication with the SC/FOX BOX takes place across a 56K-baud RS-232 serial port, to either a PC compatible computer or a stand-alone terminal. The SC/FOX BOX boots from its on-board EPROMs, and loads software or data from the SCSI hard disk, floppy drives, or serial port.

SC/Forth Language
* Interactive language, Forth-83 Standard.
* Automatic optimization and μcode support.
* Time-slice or user-defined multitasking.
* Code overlay system and program spawning.

Mini-Tower Enclosure: 13"x16"x7".
* 200-watt power supply.
* Two 5¼" half-height bays and three 3½" half-height bays (one for 3½" hard disk).
* Optional full-tower enclosure.
* Wyse 150 terminal and software.
* Custom configurations quoted.

SC/FOX SBC Single Board Computer
* Uses Harris RTX 2000tm Forth Chip.
* 8, 10, 12, or 14 MHz operation.
* 64K to 512K bytes zero-wait state SRAM.
* 56K baud serial terminal port.
* Centronic parallel printer port.

SC/FOX SCSI I/O Daughter Board
* SCSI port for disk drives, optical drives, etc.
* Floppy disk port supports up to 4 disk drives.
* 56K baud serial port.
* 16-bit bidirectional latching I/O port.

External Ports
* 9-pin RS-232 terminal and serial port.
* 25-pin Centronic parallel port.
* 40-pin bidirectional parallel port.
* 50-pin SCSI I/O port.

Standard Mini-Tower Configuration
* 8 MHz SBC with 128K bytes SRAM.
* SCSI I/O board with software drivers.
* Two 5.25" 1.2M byte floppy drives.
* One 40M-byte SCSI hard disk drive.
* SC/Forth in on-board shadow EPROM.
* Room inside for custom hardware.

Ideal for embedded real-time control, high-speed data acquisition and reduction, image or signal processing, or computation-intense applications. For additional information, please contact us at:

SILICON COMPOSERS INC   208 California Avenue, Palo Alto, CA 94306   (415) 322-8763

FORTH
DIMENSIONS

EDITORIAL

Phil Koopman, Jr. has presented us with a dose of reality therapy this month in his article, "Embedded Control as a Path to Forth Acceptance." His thoughts center not so much on embedded control as on Forth's future and our own inner motivations. Is Forth worth fighting for? Are the key players burned out? Is it someone else's responsibility? We expect to hear from readers about this one, so let us know your thoughts on the matter.

*   *   *

Then there's that old story about the cook who always cut off the ends of a roast before putting it in the oven. "Why?" the thrifty spouse inquired. That's how the cook's mother had always done it. "But why?" It turned out she had not had a pot large enough to cook the whole roast. End of tale.

Our moral of that story applies to *FD* cover dates. Your issues for this volume will be arriving earlier than they used to; the same thing happened last year, too. Someone whose identity is lost in the mists of time once decided that our issues should arrive at your door around the first of the *second* month printed on the cover. That tradition was passed down volume-to-volume. Unfortunately, it made even timely issues seem a month late. When we got a complaint about an issue that hadn't arrived, we couldn't tell if it just seemed late or if it had really gone astray somewhere. So last year we sneaked all our deadlines up by two weeks and by another two weeks this year; we will still be timely, and now it will seem that way, too. Thank you, to all our advertisers, and to the printers, mailers, and others who held up under the accelerated schedules.

Author, Author!

This is the best place I know to remind you that this magazine thrives only via the participation of its readers. *Forth Dimensions* is the primary mouthpiece of the Forth Interest Group, but we have no team of writers providing examples of "correct" Forth: just people like you, working things out in the real world and willing to share your experiences, mistakes, and discoveries with others.

Write to tell us about Forth at your work, about a utility or trick you discovered, or about doing business in the Forth world. How FIG and *Forth Dimensions* — even Forth itself—rise to meet the new decade will depend on the people who shape it today. That means being resourceful and playful, and exploring the Forth philosophy in any way that interests you; but we especially hope it will mean writing for these pages and for other publications.

Reviewers' Notes

As I have pointed out elsewhere, the review process for articles published in *Forth Dimensions* is less formal than some of the scientific or academic publications. Nonetheless, articles are evaluated for technical content and other factors, with a view to making the best use of our finite space. This time I want to share with you some of the interesting comments I received from reviewers about two items in this issue:

"'Anonymous Things' by Leonard Morgenstern. A mechanism for declaring headerless variables. A method of creating classes of objects is described, instances of which can be likened to deferred words, but not as restrictive. They can later be assigned to act like variables, executable routines, even headerless (i.e., anonymous)

(Continued on page 22.)

Forth Dimensions
Published by the
Forth Interest Group
Volume XII, Number 1
May/June 1990
Editor
Marlin Ouverson
Advertising Manager
Kent Safford
Design and Production
Berglund Graphics

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $30 per year ($42 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 8231, San Jose, California 95155. Administrative offices and advertising sales: 408-277-0668.

Copyright © 1990 by Forth Interest Group, Inc. The material contained in this periodical (but not the code) is copyrighted by the individual authors of the articles and by Forth Interest Group, Inc., respectively. Any reproduction or use of this periodical as it is compiled or the articles, except reproductions for non-commercial purposes, without the written permission of Forth Interest Group, Inc. is a violation of the Copyright Laws. Any code bearing a copyright notice, however, can be used only with permission of the copyright holder.

About the Forth Interest Group

The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. Many research hardware and software designs that will advance the general state of the art. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $24/36 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 8231, San Jose, CA 95155."

LETTERS

'Fast Thousand' Seems Slow

Dear Editor,

I was curious about the program for computing the first 1000 primes, submitted by Allan Rydberg in *FD* issue XI/5. It appears to contain some original ideas for calculating primes. It is closer to the methods that test individual numbers than it is to the classical Sieve of Eratosthenes, which operates on an array.

However, the title "A Fast Thousand Primes" seems to be a misnomer. Mr. Rydberg says his program computers the first 1000 primes in a little over two minutes on a 32-bit machine running jForth. That seems awfully slow to me.

The accompanying code is my straightforward implementation of the Sieve entirely in high-level F-PC. On my 12 MHz '286 machine, it calculates the first 1000 primes in 0.28 second. It takes a few additional seconds to print these primes to the screen. The `SIEVE` word, which calculates and counts the primes, takes 0.38 second and seems to be slightly faster than the Colburn Sieve as published in *Dr. Dobb's Journal* and included here for reference. It has been said that the Colburn Sieve and the Byte Sieve return an incorrect number for the count of primes, but that this does not affect their usefulness as benchmarks. I believe that my version of the Sieve returns the correct number of primes and does so in less time.

In *An Introduction to the History of Mathematics* by Howard Eves (5th ed., 1983), the Sieve is covered as follows:

"In arithmetic, Eratosthenes is noted for the following device, known as the Sieve, for finding all of the prime numbers less than a given number n. One writes down, in order and starting with 3, all the odd numbers less than n. The composite numbers in the sequence are

```
COMMENT;


: SQRT ( n -- n' ) \ Morely method for integer square root
       1 10 0 DO
       2DUP / + 2/
       LOOP NIP ;

 7919 VALUE NPRIME   \ the largest prime to find; the 1000th prime

 NPRIME SQRT VALUE SQRTNPRIME       \ the largest possible factor of NPRIME

 CREATE PARRAY  NPRIME 2+ ALLOT      \ create the array of proper size
                                     \ two bytes for each 16bit odd number

: FILL-ARRAY          \ this  fills the array with  odd ONLY
       NPRIME  0 DO
       I  1+          \  1 to NPRIME    2*I + 1
       I PARRAY +     \  0 to NPRIME/2
       !
       2 +LOOP
       ;

0 VALUE 2N    \ to cut down on stack shuffle

                                \ cross off every nth number
: N-CROSS     ( n -- )          \ 0 into     3N-1  +2N +2N +2N ...
       DUP 2*                   \ N 2N
       DUP =: 2N                \ N 2N
       + 1-                     \ 3N-1
       BEGIN
       DUP                      \ 3N-1 3N-1 --
     . PARRAY +                 \ 3N-1 P(3N-1) --
       0 SWAP !                 \ 3N-1 -- 0 into p(3n-1)
       2N +                     \ 3N-1 +2N --
       DUP NPRIME >             \ go till limit
       UNTIL DROP
       ;

: SIFT   \ zero the non-primes,  from 3 to square root n,  primes to n
       SQRTNPRIME 3 DO          \ 89 is squ root of 7907
       I  1- PARRAY + @         \ 3 is p(2) ... 5 is p(4)
       0 > IF                   \ was NOT IF, which was slow
            I N-CROSS
          THEN
       2 +LOOP
       2 PARRAY  ! .     \ a fudge to put 2  into the array of odds
       ;

0 VALUE COUNTS        \ the number of primes found

: PRIMES-OUT           \ go through the array and print the non-zero numbers
       0 =: COUNTS    \ along with the rank of each
       NPRIME  0 DO         \ 3 is p(2)  , 2 is p(0), the "1th" prime
          PARRAY I  + @
          DUP 0 > IF        \ do this for non-zero numbers
                    INCR> COUNTS
                    COUNTS   . ." TH PRIME= "
                    . CR
                 ELSE DROP THEN
       2  +LOOP
       ;
```

then sifted out by crossing off, from 3, every third number, then from the next remaining number, 5, every fifth number, then from the next remaining number, 7, every seventh number, then from the next remaining number, 11, every eleventh number, and so on. In the process some numbers will be crossed off more than once. All the remaining numbers, along with the number 2, constitute the list of primes less than n."

It is sometimes difficult to improve much on the work of the Ancients.

Sincerely,
Marc Hawley
P.O. Box 716
Mt. Vernon, Indiana 47620

No Provisions...
Sirs,

I have been a member of FIG for some five years now and enclose with this letter my membership renewal. However, I have become increasingly irritated by the fact that FIG makes no provision for any other computers than IBMs and Macs. I use these intensively at work. At home I have an Apple (not a Mac!).

I know it would be unreasonable to expect as much attention to Apples, Amigas, Ataris, etc. as to the Big Two. *No provision at all*, however, has made me slowly lose interest in your group or your magazine.

Hey guys, I though Forth was a language for all computers, not part of the MS-DOS, OS/2, and Mac operating systems alone.

Thomas Donaldson
Sunnyvale, California

...But Not a Wasteland
Dear Editor,

I'm writing in the hope that you may know the whereabouts of Charles E. Eaker, who wrote eFORTH for the Color Computer, distributed by Frank Hogg Laboratory in Syracuse. I'm very pleased with eFORTH, except the master disk seems to have become corrupted in the intervening years. I was politely sent another defective disk ... [and] finally received the information that they could do nothing for me.... In the event that you are unable to put me in touch with Mr. Eaker, perhaps someone who reads your "Letters" column might have a copy of the original.

Despite having heard for years that Forth is the Ferrari of programming languages, I dismissed it because there is also the perception that Forth programmers are rather fanatical. But now I know why they are, and I'm converted.

Your magazine's articles about shadow stacks, string stacks, and data stacks have been especially fascinating to me. I'm amazed by every issue of *Forth Dimensions* because it's the meatiest publication I've ever seen. It has more guts than *BYTE*, despite the difference in size.

I like another thing about *Forth Dimensions*. Unlike other computer publications which are not machine-specific, you don't give the impression that there is nothing but wasteland beyond MS-DOS. Even some (like *Unix World*) that you would expect to be impervious to that syndrome have become infected to some degree.

Donald Hicks
355 St. Emanuel St.
Mobile, Alabama 36603

```
: PRIMES-COUNT          \ go through the array and count the non-zero numbers
        0 =: COUNTS    \        with the rank of each
        NPRIME  0 DO        \ 3 is p(2)
           PARRAY I  + @
            O > IF
                  INCR> COUNTS
                THEN              \ but don't bother printing them
        2  +LOOP
        COUNTS . ." primes"     \ just print the total number found
        ;


: PRIMES    \ calculate and print out 1000 primes

        FILL-ARRAY     \ put odd numbers into the array
        SIFT           \ zero out the non-primes
        PRIMES-OUT     \ print the list of primes
        ;

: DO-PRIMES       \ calculate the primes but do not display them
                  \ for a test of speed
                  \ then use PRIMES-OUT to display them, if desired
        FILL-ARRAY
        SIFT
        ;

: SIEVE                     \ calculates primes and counts them , the benchmark
        FILL-ARRAY          \ uses 8192 for NPRIMES
        SIFT
        PRIMES-COUNT        \ I believe this algorithm returns the CORRECT
        ;                   \ number of primes, unlike the benchmark.

: 10-SIEVE                  \ for speed contests
        10 0 DO
                SIEVE
        LOOP
        ;

COMMENT:
  The following is the Colburn Sieve as published in DDJ. It is a more
  efficint version of the SAME algorithm used in the Byte benchmark Sieve.

  The Colburn Sieve  typically runs in about 60% of the time of the
  Byte Sieve .  It has been said that the fact that this algorithm returns
  the   WRONG NUMBER OF PRIMES  , does not affect its usefulness as
  as a benchmark!
COMMENT;

 decimal
 8192 constant size
 variable flags   size allot


 : do-prime.hi  ( 11 seconds for 32-bit dtc forth )
   flags  size 1  fill  ( set array )
   0 ( 0 count ) size  0
   do  flags  i + c@
     if  3 i + i + dup i + size <
        if size flags +  over i +   flags +
             do   0 i c!  dup
             +loop
        then drop  1+  ( bump prime counter)
    then
  loop
  .  ." primes "
 ;

 : 10-times.hi  10 0 do do-prime.hi loop ;
```

Editorial interlude:

Every year, it seems, someone asks if we are ignoring the users of Brand X computers. A preponderance of our articles are developed on the machines that a preponderance of our readers use, naturally, but most of those articles are not about the machines, they are about a Forth technique or innovation. Even readers using the same hardware often end up modifying the printed code unless they happen to use an identical Forth system, too.

We happily publish articles whose point of reference is Hardware horribilis, *if it is of value to enough readers; and the occasional, highly specific article does find its way into print. It is a tightrope balancing act and, by the above opposing opinions, I guess we haven't fallen off either side yet. Rate (or berate) us again after the ST articles coming out soon...*

Forth-and-Back

Dear Marlin:

When I have a problem which begs for a differently optimized Forth, I can switch to a different Forth system to help solve it. But this requires that I make myself aware of the different Forth products available for a variety of platforms. I regret the time required to evaluate various Forths and to estimate how much time converting each would take. I would be confronted with similar trials and tribulations if I were programming in any other language, but Forth could rise above the other languages in this regard through support for a variety of dictionary data structures.

Yesterday I may have needed dictionary data structures that were optimized for memory efficiency. But today I may need them to be optimized for execution speed or for dynamic memory allocation.

If Forth vendors could provide a family of colon routines and associated inner interpreters, I should be able to avoid learning about and using a wide variety of Forth systems. For example, I can imagine a `T:` definer for declaring a token-threaded word, `R:` for declaring a relocatable word, and `D:` for declaring a direct-threaded word. Each of these would establish a different compilation state. Multiple versions of the compiling words (`IF`, `THEN`, etc.) would be needed, producing multiple compiling subsystems, one for each compilation state. Within one compiling subsystem, the compiling routines would help fashion parameter fields in a consistent way.

I like the ease with which token-threaded code can be manipulated. I envision myself using token-threaded versions of selected definitions until I was sure they were debugged. Perhaps I would need to patch them substantially in the process. Thereafter, I could switch to a different compiling subsystem so that another kind of optimization could be realized. Even current Forth systems support words with differently interpreted parameter fields. Code words are interpreted correctly by Forth's address interpreter as long as the exits from them are normalized (by ending them with a jump to `NEXT`). The address interpreter also correctly interprets colon definitions and instances of data structures. So, much of the flexibility needed to support multiple compiling subsystems is already present.

However, if and when vendors supply such compiling subsystems, my selection of dictionary data structures is likely to be jeopardized: the vendor would probably determine the structure of the vocabularies, dictionary linkages, and name fields.

A Forth-and-Back language would be ideal. No dictionary would exist until it is created by the user. Such a language would be able to produce Forth compiling systems made to the personal specifications of each of its users. For example, the name fields, dictionary links, and any vocabulary provisions would be chosen by the user. Until initial definitions are created by the user, no dictionary would exist. (This does not make an initial Forth text interpreter impossible, but it requires the implementation of it to be substantially different than is customary.)

Perhaps the first step would be to create an address interpreter toolset. Other toolsets would help us extend the language into a complete Forth system. If desired, these same toolsets could be used to establish support for several compiling systems at once.

Sincerely,
Mike Elola
1055–102 N. Capitol Ave.
San Jose, California 95133

For the Analyzer Toolkit

Dear Mr. Ouverson,

I have been meaning for some time to pass on to the FIG membership my most heavily used Forth word, which I call `DO-FOREVER`. Its function is to execute the next input word repeatedly until a key is pressed. I use fig-FORTH to write automatic test programs for bus-controlled instrument cards. `DO-FOREVER` (see Figure One) allows me to repeat a dubious bus transaction rapidly enough to follow what is happening with an oscilloscope.

The word that follows `DO-FOREVER` may be one of the existing words in the test program, but is more often a word written at the keyboard to check the problem I am currently facing. For example, if a control signal were not having the desired effect, I would write a definition to toggle a control line every five milliseconds (see the same figure), then follow the signal through the board with a scope probe until I found where it got lost. `DO-FOREVER` is now a fixture in all my automatic test programs.

Best wishes,
Tom Napier
One Lower State Road
North Wales, Pennsylvania 19454

```
: DO-FOREVER
  ( re-execute next word until keypress )
  -FIND  IF DROP CFA  ( this is fig-FORTH )
  BEGIN
    DUP EXECUTE ?TERMINAL
  UNTIL DROP  ;

: TEST
  0 7 >REG   ( clear register 7 )
  5 MILLISEC ( wait )
  4 7 >REG   ( set bit 2 of register )
  5 MILLISEC  ;

DO-FOREVER TEST  ( toggle bit 2 until keypress )
```

Figure One. Napier's `DO-FOREVER` and a typical test word.

ANONYMOUS "THINGS"

LEONARD MORGENSTERN - MORAGA, CALIFORNIA

Several years ago, I proposed a scheme of anonymous variables, that is, headerless variables used as temporaries.[1] In this article, the idea is extended to other classes of Forth words, including colon and code definitions. Anonymous "things" have three advantages: First, they save memory by eliminating headers. Second, the programmer is spared the burden of inventing unique names for words used only once or twice. Third, anonymous words reduce, although they do not eliminate, the risk of error from duplication of names. They are most useful when a program is compiled in segments, or when large execution arrays consume many colon definitions.

In the system presented here, illustrated in F83, it is possible to create anonymous "things," assign them temporary names, and use them to pass information back and forth among a group of consecutively defined Forth words, not only with respect to data (variables and constants), but also program (colon definitions). There is a similarity to the locals used in Pascal and certain other languages, but there is no provision for dynamic allocation of memory at run time, essential for large temporary arrays. The resemblance to Pascal can be made closer by adding automatic nesting of scope, not further discussed here.[2]

Creating and Using Anonymous Things

Words that create anonymous things do so by laying down a code field and body, omitting header information. The address of the code field is left on the stack, ready to insert into an execution vector.

1. "Anonymnous Variables," Forth Dimensions VI/1.
2. See proceedings of ACM's SIGForth, February 1989 meeting.

An analysis of `NCONSTANT` illustrates the principle underlying anonymous definitions.

`: NCONSTANT (n -- adr)`	Start the definition. As with an ordinary constant, the value to be stored must be on top of the stack.
`PSEUDONAME`	If in debugging mode, lay down a pseudoname; other wise, no action.
`HERE >R`	The CFA of the word will be at `HERE`. Put that address on the stack, then hide it on the return stack.
`['] B/BUF @ ,`	Compile the contents of the CFA of an existing constant, namely `B/BUF`.
`,`	Compile the body of the anonymous constant. The action here duplicates the compile-time action of `CON STANT`, which is to "comma" the value on top of the stack.
`R> ;`	Bring back the CFA from the return stack and leave it as a hook for later action.

Table One. Analysis of `NCONSTANT`.

```
IMPERSONATOR TEMP
N: ." Hello." ;  IS TEMP
: FOO1 TEMP ;
```

An anonymous colon definition is assigned to the impersonator `TEMP` which becomes, in effect, an ordinary colon definition, whether compiling or interpreting. Thus, executing `TEMP` will display "Hello." and `FOO1` will do the same.

```
NVARIABLE IS TEMP
: FOO2  TEMP ? ;
```

`TEMP` is now a variable. `FOO2` will display its contents. `FOO1` will still display "Hello."

```
' + IS TEMP
: FOO3  3 3 TEMP . ;
```

`TEMP` is now an alias for +. `FOO3` will display the number six.

Table Two. Using `IMPERSONATOR`.

Screen 2 (Required)

```
VARIABLE DEBUGGING      DEBUGGING OFF
: PSEUDONAME  DEBUGGING @  IF 129 C, 238 C, THEN ; IMMEDIATE

: NVARIABLE  ( ... addr)
PSEUDONAME  HERE  [']  >IN @ ,  0 , ;
: NCONSTANT ( n ... addr)
      PSEUDONAME HERE >R ['] B/BUF @ ,     , R> ;
: IMPERSONATOR    CREATE [']  CRASH ,      IMMEDIATE
      DOES> @     STATE @     IF ,  ELSE EXECUTE  THEN ;

IMPERSONATOR      TEMPC
IMPERSONATOR      TEMPD
NVARIABLE IS TEMPC      \ TEMPC now acts as a variable
: N:  PSEUDONAME HERE TEMPC ON
      ['] NVARIABLE @ ,  !CSP CURRENT @ CONTEXT !  ] ;

: :  TEMPC OFF [COMPILE] : ;
: ;  TEMPC @  IF ?CSP COMPILE UNNEST [COMPILE] [  TEMPC OFF
     ELSE [COMPILE] ;  THEN ; IMMEDIATE
```

Screen 3 (Optional)

```
: NCODE     PSEUDONAME HERE DUP 2+ ,
            CONTEXT @ AVOC ! ASSEMBLER ;

\ Ancillaries
: (NCREATE)  PSEUDONAME HERE 2 ALLOT ;
: N(;CODE)  R> OVER ! ;
: (NDOES>)  COMPILE N(;CODE)
            232 C, 333 HERE 2+ - , ; IMMEDIATE

\ NCREATE and NDOES>
: NCREATE   COMPILE (NCREATE) COMPILE >R ; IMMEDIATE

: NDOES>    COMPILE R> [COMPILE] (NDOES>) ; IMMEDIATE
```

Screen 4 (Example: Defining N2CONSTANT by Method 1)

```
\ Step 1. Define a 2CONSTANT
0. 2CONSTANT ZERO

\ Step 2. Use the defined 2CONSTANT to define others.
: N2CONSTANT   ( -- addr) PSEUDONAME
      HERE >R  ['] ZERO @  , , ,  R> ;
```

I have selected the letter N, standing for "nameless," to be the indicator of an anonymous definition. The letter A, for "anonymous," was not available, because `A:` is a previously defined word in F83. Definitions of the most frequently used types, `NVARIABLE`, `NCONSTANT`, and `N:`, are on screen two, and `NCODE` is on screen three.

For example, to create an anonymous constant with the value three, type `3 NCONSTANT`. (Do not try to give it a name—remember, it's anonymous.) An anonymous variable is created by simply writing `NVARIABLE`, an anonymous colon definition by the pair `N: ... ;`, and an anonymous code definition by `NCODE ... END-CODE`.

An anonymous thing is not useful until its CFA is assigned to some kind of execution vector. One could, for example, insert it into a deferred word. More serviceable is `IMPERSONATOR` (screen two), which differs from `DEFER` in that it compiles its contents at compile time, whereas a deferred word compiles itself. One assigns a word to an impersonator the same as to a deferred word, by means of `IS` or `[IS]`. The assigned action does not have to be anonymous, nor does it have to be a particular type. Table Two shows examples. Anything compiled by an impersonator is unaltered by a new assignment. An impersonator acts like an ordinary definition in most respects, but will not respond in the usual way to words that forward-reference the input stream, such as `'` and `[']`. Also, if the assigned word is immediate, it will be compiled regardless.

The number of impersonators is determined by how many assignments have to be in effect at the same time, not by how many are made in the program as a whole. As a rule, three or four are plenty, but extras may improve readability. Several cautions are in order: The system should not be overused; a word present on more than two or three consecutive screens should be named. Also, due care is needed when the scopes of anonymous "things" overlap.

Creating a New Class of Anonymous Things

In practice, I find that `NVARIABLE`, `NCONSTANT`, and `N:` are enough anonymous classes, and `NCODE` is occasionally useful, too. To make more, two techniques are available. The first, or basic, method was used to create the above definitions, and is illustrated further in Table One and Screen Four. It is the method of choice when a named class already exists. One makes a named example, and uses it as a basis for creating the anonymous class. If the corresponding named class does not exist, one can simply create a definer for it in the ordinary way. In that case, the new named and anonymous definers will have identical "create" parts, so the common code should be factored out into an anonymous colon definition, as in the example on Screen Five.

A better, direct, way to create a new anonymous class is provided by the second method, which employs the pair NCREATE ... NDOES>, defined on Screen Three and illustrated on Screen Six. No named definer is necessary, but the latter can be created at the same time, if desired.

Pseudonames for Debugging

Since anonymous things do not have a name field, the display that results from DEBUG and SEE is likely to be unreadable. To circumvent this, the word PSEUDONAME has been inserted in all definitions. Its action depends on the contents of an ordinary variable named DEBUGGING. If the latter contains TRUE, anonymous things will be compiled with a pseudoname field consisting of the two bytes 129 and 238, corresponding to lower case n. It is not a true header, as no link or view fields are laid down. Appropriate use of the phrases DEBUGGING OFF and DEBUGGING ON will assist program development.

Explanation of Source Screens

Only Screen Two is required. It contains the essential definitions, including DEBUGGING, PSEUDONAME, NVARIABLE, NCONSTANT, N:, and IMPERSONATOR. Redefinition of : and ; is necessary. Two impersonators, named TEMPC and TEMPD, are also defined. The names suggest their intended use in the creation of new classes, but they are not restricted to that function. In fact, TEMPC is used as a temporary during the definition of N: and the redefinition of : (Screen Two).

Screen Three is optional. It contains NCODE, NCREATE, NDOES>, and their ancillaries.

Screens Four through Six contain ex-

(Continued on page 37.)

Screen 5 **(Defining 3CONSTANT and N3CONSTANT by Method 1)**

```
\ Step 1. Define the "Create" part as an anonymous.
\ Assign it to TEMPC
N:  , , , ; IS TEMPC

\ Step 2. Define a normal creator, using
\ TEMPC in its "create" part
: 3CONSTANT  CREATE TEMPC
     DOES> DUP 2+ 2@ ROT @ ;


\ Step 3. Create an example.
0 0 0 3CONSTANT 3ZEROES

\ Step 4. Use the example as a basis for the
\ anonymous creator.
: N3CONSTANT  PSEUDONAME
  HERE >R ['] 3ZEROES @ , TEMPC R> ;

\ Testing. Note that TEMPC can be reused without conflict
CR .( SHOULD PRINT 0 0 0 )    3ZEROES . . . CR
1 2 3 N3CONSTANT IS TEMPC
CR .( SHOULD PRINT 3 2 1 )    TEMPC . . . CR
```

Screen 6 **(Defining 4CONSTANT and N4CONSTANT by Method 2)**

```
\ If only N4CONSTANT is needed, simply write
: N4CONSTANT      NCREATE     2SWAP , , , ,
                  NDOES>      DUP 2@ ROT 4 + 2@  ;

\ If both 4CONSTANT and N4CONSTANT are needed,
\ there are 2 steps:
\ 1. Write the CREATE and DOES> parts;
\ assign them to impersonators
N: ( n4 n3 n2 n1 -- )
   2SWAP , , , ,  ;            IS TEMPC
N: ( adr -- n4 n3 n2 n1)
   DUP 2@ ROT 4 + 2@  ; IS TEMPD

\ 2. Write the definers according to the
\ following set formula:
: 4CONSTANT CREATE   TEMPC    DOES>   TEMPD ;
: N4CONSTANT      NCREATE TEMPC     NDOES> TEMPD ;

\ The impersonators are now free for reassignment.
```

INTERPRETATION-MACRO TECHNIQUES

CHESTER H. PAGE - SILVER SPRING, MARYLAND

All computers provide input/output routines for disk files; some Forth dialects ignore this and substitute their own concepts of buffers and file management. This always impressed me as an attempt to make Forth appear independent and self-sufficient, but the read/write commands had to be tailored to the host computer anyway, and were rarely any improvement over the host DOS.

I had not used Forth long before I decided that the way to implement disk I/O was to develop communication links between Forth and the non-Forth procedures provided by the host DOS. Some of the procedures suggested similar operations for use within Forth. This paper describes some of these techniques.

Synopsis

Two types of interpretation macros can be very handy. The first type is an input-stream macro: text strings are accumulated in some buffer area and then made an input stream for interpretation or compilation. The second type is a synthetic keyboard input to provide a link with the non-Forth world; specifically, this provides access to host computer operating system routines for cataloging a diskette, sending output to a printer, loading a text file from a diskette, saving a file to a diskette, etc.

First Type Used in Interpretation

Sample use: to avoid repetitive keyboard entries

In developing a matrix inversion program for my `FLOAT.FORTH`, I encountered the following situation. To use the routines for inverting an n x n matrix, I needed to define four n x n matrices called A, B, E, and X, and then save their code-field addresses in variables `'A`, `'B`, `'E`, and `'X`. This required entering:

```
n n FMATRIX A
n n FMATRIX B
n n FMATRIX E
n n FMATRIX X
' A 'A !
' B 'B !
' E 'E !
' X 'X !
```

which is rather tedious.

This ... can be made specific to any operating system.

I therefore developed a word such that entering n `MAKESET` would carry out all these operations.

Assuming the existence of the `FMATRIX` defining word and the modification of `INTERPRET` to `FINTERPRET`, the only special word needed is `>TIB` (discussed later):

```
: MAKEREF
  >TIB ' A 'A !
       ' B 'B !
       ' E 'E !
       ' X 'X !
  !" 0 DUP >IN
  BLK ! FINTERPRET  ;

: MAKESET  ( n -- )
  DUP DUP DUP DUP
  DUP DUP DUP
  >TIB FMATRIX A
       FMATRIX B
       FMATRIX E
       FMATRIX X"
  0 DUP >IN !
  BLK ! FINTERPRET
  MAKEREF  ;
```

(Note the " delimiter on the string to be sent to `TIB` in both these words.)

It is apparent that entering n `MAKESET` is intended to replace the original sequence of eight commands.

The key to all this is in `>TIB`, whose definition is similar to that of `."`:

```
: >TIB
  ?COMP  34 COMPILE
  (>TIB)  WORD C@ 1+
  ALLOT  ; IMMEDIATE
```

With:

```
: (>TIB)
  PHRASE SWAP OVER
  TIB #TIB @ + SWAP
  CMOVE #TIB +!
  0 TIB #TIB
  @ + !  ;
```

After moving text to `TIB`, this enters an end-of-stream null.

Note the new word `PHRASE`. It resets the next-word pointer to beyond in-line text compiled by `WORD` as a dimensioned string.

```
: PHRASE ( -- addr +n )
  R> R@ COUNT
  DUP 1+ R>
  + >R  ROT >R  ;
```

First Type Used in Compilation

Consider the trivial word definition:

```
: TEST
  7 1 DO
  I .   LOOP   ;
```

I shall use this as an example, using `>TIB`. First, the obvious incorporation of this definition directly into a macro:

```
: MAKE.TEST
  0 #TIB !
  >TIB
    : TEST 7 1 DO
      I . LOOP ; "
  0 DUP >IN !
  BLK ! INTERPRET   ;
```

Now let's break it up into pieces:

```
: (A) >TIB : TEST "   ;
: (B) >TIB 7 1 DO "   ;
: (C) >TIB I . "  ;
: (D) >TIB LOOP ; "   ;
```

and define:

```
: MAKE.TEST
  (A)  (B)  (C)  (D)
  0 DUP >IN !
  BLK ! INTERPRET   ;
```

This loads the terminal input buffer piece-wise at `TIB`, then interprets the complete load.

Entering `MAKE.TEST` either from the keyboard or a screen will generate `TEST`.

Expansion to Long Strings

The above technique is limited by the available space in `TIB`. Expansion to commands of a kilobyte can be made by storing the potential input stream in the massbuffer as a dummy *screen #-1*. A word `ADD.BUF` is used to append text to what is already in the buffer, using `BUF.POS` as a position pointer. (Details of the following procedure can depend on the screen-input system of your Forth dialect—mine uses a single 1024-byte massbuffer.)

```
VARIABLE BUF.POS
: APPEND.BUF ( addr +n -- )
  BUF.POS @ MASSBUF
  + SWAP DUP
  BUF.POS +! CMOVE
  0 MASSBUF BUF.POS
  @ + !   ;
```

The last part adds two nulls at the end of the text; these nulls will be overwritten by the next `APPEND.BUF`.

```
: (ADD.BUF)
  \ Run-time routine
  PHRASE APPEND.BUF   ;

: ADD.BUF
  34 STATE @
  IF COMPILE (ADD.BUF)
    WORD C@   1+ ALLOT
  ELSE WORD COUNT
    APPEND.BUF THEN
  ;  IMMEDIATE
```

Use by entering:
`0 BUF.POS ! ADD.BUF`
<first batch of strings,
not exceeding 256 bytes>
`ADD.BUF` <second batch>
`ADD.BUF` <third batch> ...

After the buffer is loaded, `-1 LIST` will display it; `-1 LOAD` will interpret it.

To illustrate, in the fragments (A) - (D) above, replace `>TIB` with `ADD.BUF` and initialize the massbuffer instead of `TIB`:

```
: (1) ADD.BUF
      : TEST "   ;
: (2) ADD.BUF
      7 1 DO "   ;
: (3) ADD.BUF
      I . "   ;
: (4) ADD.BUF
      LOOP ; "   ;

: MAKE.TEST
  0 BUF.POS  !
  -1 BUF.ID  !
  (1)  (2)  (3)  (4)   ;
```

From the *keyboard*, enter `MAKE.TEST -1 LOAD` to generate `TEST`.

Warning: do *not* enter `MAKE.TEST -1 LOAD` from a screen, because `ADD.BUF` modifies the massbuffer and could mess up the screen being used, via a flush.

Second Type: Link to Host DOS

Under ProDOS, the Apple][accepts many direct keyboard commands, such as `CATALOG`, `DELETE <file>`, `PR#1`, `BSAVE`, `BLOAD`, etc. The keyboard input

BRODIE

EDUCATIONAL SERVICES

Technical and User Manuals
Courseware
User Interface Design

Leo Brodie
1530 Eland Lane
Ventura, CA 93003
805/647-7315

is checked by a command parser in the BASIC Interpreter ROM. With a special word, DOS, these operations can be accessed from the keyboard by entering:

```
DOS CATALOG
DOS DELETE <file>
DOS BLOAD <file>
```

Or, from Forth words:

```
: CATALOG
  DOS CATALOG"  ;

: FILEOUT
  DOS DELETE <file>"  ;

: RUN-PROGRAM
  DOS -PROGRAM"  ;
```

These "one-piece" macros are based on a primitive DOS.CALL which, after the desired text is moved to the keyboard buffer, calls the built-in command parser using:

```
: >KEYBUF ( addr +n -- )
  KEYBUF ROT ROT
  ?DUP IF
    OVER + SWAP
      DO I C@
      128 OR OVER C!
      1+ LOOP
    141 SWAP C!
  ELSE
    WORD COUNT
    >KEYBUF DOS.CALL
  THEN  ; IMMEDIATE
```

This "synthetic keyboard input" can be made specific to any computer operating system by appropriately defining the primitive DOS.CALL.

More elaborate DOS commands require piecewise build-up. For example, I save a Forth application by entering SELFSAVE, which requests a filename, then automatically opens such a file and BSAVEs from the origin of the Forth program to its end, automatically setting the FENCE.

The routine is based on concatenating strings in a "buffer" which is the parameter space of a variable named COMMAND, created by

```
VARIABLE COMMAND
100 ALLOT
```

(or whatever size is desired). The file-name—more generally, the pathname—is held in a "string" location:

```
VARIABLE STRING
64 ALLOT
```

(to allow for the maximum legal ProDOS pathname length). The phrase:

```
STRING 1+
64 EXPECT
SPAN @ STRING C!
```

is used to load the pathname.

The string in COMMAND is converted to a DOS command by:

```
: GODOS
  COMMAND COUNT
  >KEYBUF DOS.CALL  ;
```

Strings are concatenated to the string in COMMAND by:

```
: CONCAT   ( addr +n -- )
  DUP COMMAND DUP
  C@ + 1+ SWAP
  COMMAND +! SWAP CMOVE  ;

: (ADD$)
  PHRASE CONCAT  ;

: ADD$
  34 STATE @ IF
    COMPILE (ADD$) WORD
    C@ 1+ ALLOT
  ELSE
    WORD COUNT CONCAT
  THEN  ; IMMEDIATE
```

and a special word to concatenate the file pathname:

```
: ADD.STRING
  STRING COUNT CONCAT ;
```

Numbers are incorporated by:

```
: DIGITIZE ( u -- )
  0 <# #S #> CONCAT  ;
```

which converts the number on the stack to its ASCII representation and concatenates the ASCII string to COMMAND.

With these words, build SELFSAVE:

```
: SELFSAVE
  HERE FENCE !
  CR ." Enter filename"
    STRING 1+  64 EXPECT
    SPAN @ STRING C!
    0 COMMAND !
  ADD$ CREATE"
    ADD.STRING GODOS
    0 COMMAND !
  ADD$ DELETE"
    ADD.STRING GODOS
    0 COMMAND !
  ADD$ BSAVE"
    ADD.STRING
  ADD$ ,A"
    ORIGIN DIGITIZE
  ADD$ ,E"
    FENCE @ DIGITIZE
  GODOS  ;
```

SELFSAVE automatically supplies the A and E parameters required by BSAVE (ProDOS binary-file save). The CREATE-DELETE pair takes care of the situation where the file already exists but is longer than needed.

First Type, Again in Compilation

The words DOS and ADD$ used above contain the sequences:

```
(1)   34 STATE @
      IF COMPILE
(2)   WORD C@ 1+ ALLOT
      ELSE WORD COUNT
```

The sequences (1) and (2) could be defined by:

```
: (1) ADD.BUF
  34 STATE @
  IF COMPILE " ;

: (2) ADD.BUF
  WORD C@ 1+ ALLOT
  ELSE WORD COUNT
```

allowing the definition-makers:

```
: MAKE.ADD$
  0 BUF.POS !
  -1 BUF.ID !
  ADD.BUF : ADD$ " (1)
  ADD.BUF (ADD$) " (2)
  ADD.BUF CONCAT THEN ; "
  ADD.BUF IMMEDIATE"  ;

: MADE.DOS
   0 BUF.POS !
  -1 BUF.ID !
  ADD.BUF : DOS " (1)
  ADD.BUF (DOS) " (2)
  ADD.BUF >KEYBUF
    DOS.CALL THEN ; "
  ADD.BUF IMMEDIATE"  ;
```

which are to be used by keyboarding MAKE.ADD$ -1 LOAD and MAKE.DOS

(Continued on page 37.)

Eight Examples of

POSITIVE-DIVISOR FLOORED DIVISION

ROBERT BERKEY - FREMONT, CALIFORNIA

Continuity—of the quotient at zero—is one of the properties of floored division that makes it useful. The useful corollary with remainders is that the remainder of a floored division is a modulus. Floored division is one of a number of division algorithms that exhibit these characteristics.

This paper is a response to the request of several Forth programmers for more (and/or "real") division examples using floored division, and the paper also includes discussion of the relative merits of the symmetrical* division algorithm. In reviewing readily available Forth divisions, I've found a number that relate to floored division. Each of the following examples is based on code presently on-line and operational on an AT. Most of the examples are from a commercial application. In some examples, irrelevant context has been deleted for clarity.

Each case requires analysis, and ... there is little room for error.

Note: A $ prefix on a number signifies hex and a # prefix on a number means decimal.

\ Example 1

The first example is of the essence. The concept is that a `2/`, an arithmetic right shift, is floored. To help give it perspective, it's stated as a vignette.

You have a fixed-bid consulting job to speed up an unfamiliar system. You see a `2 /` in a promising location. What do you do?

Figure One.

```
: COMPUTE_LIMITS    ( normative-value -- limit1 limit2 )
\ If the normative value is negative, limit1 is the high limit.
\ If the normative value is positive, limit2 is the high limit.
\ Limits are outside the acceptable range.
  DUP >R DUP 3 / DUP >R - R> R> SWAP + ;


: DO_COMPARISON    ( limit1 limit2 measurement -- bad-flag )
  DUP ROT SWAP   DUP 0< NOT ( set up for compare )
    IF    >    -ROT    <
    ELSE    <    -ROT    >
    THEN
  AND NOT ;
```

Figure Two.

```
: COMPUTE_LIMITS ( normative-value -- lo-limit hi-limit )
  \ Limits are outside the acceptable range.
  DUP ABS 3 U/   2DUP -   -ROT + ;

: DO_COMPARISON ( lo-limit hi-limit measurement -- bad-flag )
  TUCK > -ROT < AND NOT ;
```

Figure Three.

```
: COMPUTE_LIMITS    ( normative-value -- lo-limit hi-limit )
  \ Limits are outside the acceptable range.
  DUP 3 / ABS    2DUP -    -ROT + ;
```

Figure Four.

```
\——— Extract from F-PC 3.5   PRINTING.SEQ———

6 constant pitems   \ number of printer-menu items
variable pitem   \ current printer-menu item

: prup           ( c1 -- c1 )
  pitem @ pitems 1- + pitems mod pitem !

\ Using signed code the phrase shortens:
\ pitem @ 1- pitems mod pitem !
```

* "Symmetrical." Various terms have been used to indicate this division algorithm. The Forth-79 Standard describes quotients as "rounded toward zero." Intel's

Figure Five.

```
\ Block# 3
( TABLE SINES         Trigonometry          82z09 )  DECIMAL

: TABLE   CREATE   DOES> OVER + + @ ;

TABLE SINES
0 ,  0286 ,  0571 ,  0856 ,  1143 ,  1428 ,  1712 ,  1997 ,
2280 ,  2562 ,  2844 ,  3126 ,  3406 ,  3686 ,  3963 ,  4240 ,
4515 ,  4790 ,  5062 ,  5334 ,  5603 ,  5872 ,  6137 ,  6401 ,
6663 ,  6923 ,  7182 ,  7438 ,  7692 ,  7942 ,  8192 ,  8437 ,
8681 ,  8922 ,  9161 ,  9397 ,  9630 ,  9859 , 10087 , 10310 ,
10531 , 10749 , 10962 , 11173 , 11381 , 11585 , 11785 , 11983 ,
12174 , 12365 , 12550 , 12732 , 12910 , 13084 , 13254 , 13421 ,
13582 , 13741 , 13893 , 14044 , 14188 , 14329 , 14465 , 14598 ,
14725 , 14848 , 14966 , 15081 , 15191 , 15296 , 15396 , 15491 ,
15582 , 15668 , 15749 , 15825 , 15897 , 15964 , 16025 , 16082 ,
16134 , 16182 , 16225 , 16261 , 16293 , 16321 , 16344 , 16361 ,
16374 , 16380 , 16384 ,


\ Block# 4 ( sin* cos*         Trigonometry       860201z ) DECIMAL

: (SIN)    ( N1 -- N2 )
  DUP 90 > IF   180 SWAP -   THEN   SINES ;

: SIN    ( N1 -- N2 )
  360 MOD    DUP 0< IF    360 +    THEN
DUP 180 > IF   180 - (SIN) NEGATE   ELSE   (SIN)   THEN ;
```

Figure Six.

```
: SIN    ( n1 -- n2 )
  90 /MOD TUCK
  1 AND IF    ( quadrants 1 and 3 )    NEGATE 90 +    THEN
  SINES
  SWAP 2 AND IF    ( quadrants 2 and 3 )    NEGATE    THEN ;
```

a) Forth-79

Since Forth-79 did not define 2/, you check—and find that the system has a conventional 2/.

What do you do then?

1) Analyze all inputs to the operation and verify that none are negative and odd. Change the 2 / to 2/. Or,
2) Implement a code version of a symmetrical 2/. Select a name for it and install that routine. Or,
3) You're willing to take the risk that there are no negative and odd inputs. Change the 2 / to 2/. Or,
4) Leave it alone until you've gotten more information to verify that changing it will have an impact.

b) Forth-83

1) Change the 2 / to 2/.

I'm not on a fixed-bid consulting job at the moment, but this is a problem I have right now, because in the Forth-79 application I'm working with I can't easily gain both speed and space optimization by just changing the 2 / to 2/. Each case requires analysis, and the tradeoffs are such that there is little room for error in modifications.

\ Example 2

You've been requested to add eight-bit characters for an Israeli customer. Looking at the files you see a half-dozen places with the code:

```
@ $100 /MOD EMIT EMIT
```

However, you have a Forth-79 system. What would you do?

This isn't fiction. The point here is that with a Forth-83 division it wouldn't matter (assuming that the EMITs ignore the high byte—and those in this application do just that).

Another point is that I don't think it's all that obvious that there is a problem. Even knowing of the problem, I needed some examples and some mental exercises to visualize the bits.

The principle working here is that when floored division is used to shift a negative number, the bits being shifted aren't disturbed.

\ Example 3

Here's how to make an easy rounding job hard. The DUP DROP and 0 + appear to be a history of problems with this routine.

```
: ROUND    ( n -- n )
  \ round number using LSD
  DUP DUP ABS / SWAP ABS
  #10 /MOD    SWAP DUP 5 >
    IF    DROP 1+ 0
    ELSE    DROP 0
    THEN
  SWAP    #10 *    + * ;
```

There is an interesting point here, in that this routine requires division by zero to do something. Anything. Any answer at all is ok. The only problem occurs if the system quits.

Here is this same routine in Forth-83:

```
: ROUND    ( n1 -- n2 )
  DUP 0< -
  \ bias .5 cases toward zero
  4 +    DUP #10 MOD -   ;
```

The bias of the .5 cases toward zero is questionable. In this application, it slightly distorts a graph across a zero line. The presence of floored division makes convenient a flat rounded-to-nearest behavior (rounded-to-nearest with continuity). The preferred routine here, using Forth-83, is:

```
: ROUND    ( n1 -- n2 )
  4 +    DUP #10 MOD -   ;
```

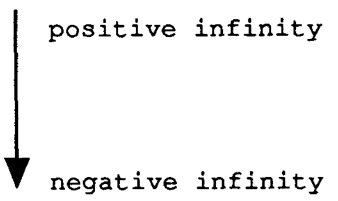
\ Example 4

Here's a very similar routine, but at a different place in the code and written by a different engineer:

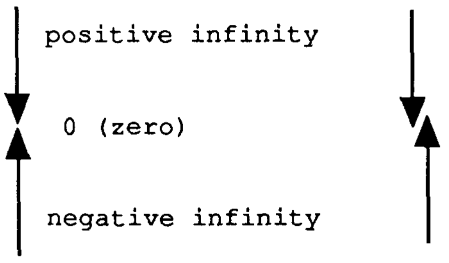
```
DUP ABS 5 +
#10 /    SWAP 0<
IF    NEGATE    THEN
```

Relationships of
Floored and Symmetrical Integer Division

If a picture is worth a thousand words, the following helps understand the relationships between floored and symmetrical division. In pictorial form, the rounding of floored division is a single arrow.



A pictorial form of the rounding of symmetrical division uses two arrows (shown two ways).



The first of these two pictures is somewhat misleading. The second picture clarifies a problem area in routines using symmetrical division. I'll get back to the overlapped arrowheads in the second picture. The initial point considered here is the number of rounding directions.

It is a given that two rounding directions is more complex than one. An example of where this complexity shows itself is in an implementation of a rounded-to-nearest division, as follows. Note that there exist several other varieties of rounded-to-nearest division algorithms—this one rounds .5 to its ceiling.

First, consider using unsigned numbers and unsigned division. Suppose that we wish to divide by ten and round to the nearest.

```
: 10u/r   ( u1 -- u2 )
  5 + 10 u/ ;
```

With floored division, this formula needs no change for signed numbers.

```
: 10/r   ( n1 -- n2 )
  5 + 10 / ;
```

With symmetrical division, the multiple rounding directions need reverse engineering.

```
: 10/r   ( n1 -- n2 )
  dup 0< if  4 -
    else  5 +
  then  10 / ;
```

Examples for all of the routines:

```
74 10u/r . 7  ok
75 10u/r . 8  ok
76 10u/r . 8  ok
```

Examples for either of the signed routines:

```
-100 74 + 10/r dup
  . 10 + . -3 7 ok

-100 75 + 10/r dup
  . 10 + . -2 8 ok

-100 76 + 10/r dup
  . 10 + . -2 8 ok
```

In addition to being more complex in the way that it rounds, symmetrical division has a structural defect. This defect is that the two arrows, as shown above, overlap at zero—either direction of rounding can produce a zero. What follows is difficult to appreciate, as demonstrated by published mistakes using symmetrical division. The quotient of a negative number divided by a positive, tested with 0<, can produce a false—because the result might be zero. This is a contradiction of the basic principles we are taught concerning signs with division and multiplication, in which a negative operated on by a positive results in a negative.

Relatedly, because of the overlap, information is lost in the symmetrical transformation. A quotient of zero represents two units of information, the numbers from greater than -1 to less than 1, while all other quotients represent one unit of information. This is why the quotient and remainder of a floored division can be converted to a symmetrical quotient, but not vice versa. Symmetrical conversion requires retaining the divisor until after the division.

Forth-83:

```
DUP 0< +
\ bias .5 cases away from zero
5 +   #10 /
```

Again, the bias of the .5 cases is dubious, and floored division makes convenient a rounded-to-nearest implementation with continuity:

```
5 +   #10 /
```

\ Example 5

The Forth-79 example in Figure One is quite different from the others. Here, floored division is clearly less appropriate than the symmetrical division chosen; yet this is useful because the pick of division here is unsigned. This makes the second case I've seen in which floored division was obviously dispreferred, while symmetrical division led to problems.

Figure Two is a restatement of the coding. Note that the `U/` is the unsigned equivalent of `/`. A principle here is that the complexity of symmetrical division can be transformed and passed along in altered form.

Another Coding.
With Forth-79, the restatement of the `COMPUTE_LIMITS` routine could be coded as in Figure Three, which is a dead heat with the unsigned routine, in terms of bytes.

First point: Once started on it, the engineer who wrote the above example stayed with a path that used signs.

Second point: The `3 / ABS` is more complicated, both in mental terms and the hardware needed to execute it, than is the `ABS 3 U/`.

\ Example 6

Figure Four is an example from the F-PC world of using a signed `MOD`. Here, we watch Tom Zimmer avoid signed division. I ran F-PC for a month with a trap in it that announced negative operators in a division or shift. The most frequent occurrence was the `2/` in `DEPTH` during stack underflow. From what I saw, the F-PC compiler could strip out signed division. That doesn't mean it couldn't benefit from using more signed division. It seems to be the pattern, though, that compilers need signed division less than applications.

\ Example 7

Code extract from the *1986 FORML Conference Proceedings* paper, "Turtles Explore Floored Division" by Zafar Essak, M.D. is given in Figure Five. Note the reverse engineering used here for symmetrical division—with the use of the modulus, the clause:

```
DUP 0< IF 360 + THEN
```

can be deleted.

Figure Six is another approach to this implementation, which happens to use both a modulus and a floored quotient—it is the shortest routine I found. Because of the reflections among the four quadrants of a sine curve, this routine is a candidate for a useful symmetrical division, but I looked and couldn't find one better than this floored routine.

\ Example 8

This final and unusual example goes beyond floored division into the realm of division theory. Note that the `U/MOD` here is the unsigned equivalent of `/MOD`.

```
: T/    ( n -- index weight )
  #100 SWAP -
  0 MAX   #299 MIN
  #100 U/MOD
  #100 ROT - SWAP ;
```

Part of what this is doing is returning remainders in the set {1..100}.

Using Forth-83:
```
: T/    ( n -- index weight )
  #-199 MAX   #100 MIN
  1- #100 /MOD
  SWAP 1+ SWAP NEGATE ;
```

The floored division shortened and simplified the solution, but the solution gets still simpler were we to implement a division algorithm that, with positive divisors, returns one-based remainders rather than zero-based remainders. With this algorithm, a division by three would return remainders in the set {1..3}. Note that this won't contradict the "division transformation"—as usual, the divisor times the quo-

(Continued on page 34.)

Total control with *LMI FORTH*™

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

For Development:
Interactive Forth-83 Interpreter/Compilers
for MS-DOS, OS/2, and the 80386
- 16-bit and 32-bit implementations
- Full screen editor and assembler
- Uses standard operating system files
- 500 page manual written in plain English
- Support for graphics,floating point, native code generation

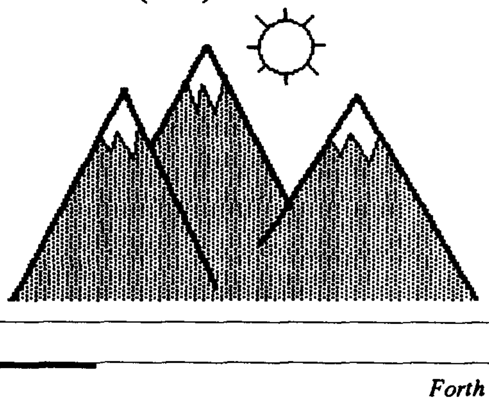
For Applications: Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 8088, 68000, 6502, 8051, 8096, 1802, 6303, 6809, 68HC11,34010, V25, RTX-2000
- No license fee or royalty for compiled applications

Laboratory Microsystems Incorporated
Post Office Box 10430, Marina del Rey, CA 90295
Phone Credit Card Orders to: (213) 306-7412
FAX: (213) 301-0761

CONSULTANTS

CIBER, a national consulting firm, has Forth assignments in the Denver area. If you are looking for a change, and the Rocky Mountains appeal to you, please give us a call or send your resume to:

CIBER
Recruiter
4100 E. Mississippi Ave., Suite 1810
Denver, CO 80222
(303) 691-2273



STACK VARIABLES

GIORGIO KOURTIS - GENOA, ITALY

Much has already been said regarding use of the stack as opposed to variables, and concerning the errors that arise when variables are included in routines called recursively.

Still, those types of errors continue to pop up unexpectedly. To illustrate, consider a word that must cope with a variable number of arguments; let's suppose a routine is needed called ADD that adds the values of a specified number of arguments, e.g.:

```
14 32 -17
3 ADD

15 79 42 76
4 ADD
```

ADD can be defined as:

```
: ADD
  ( M[n-1] M[n-2] ... )
  ( ... M[1] M[0] n -- sum )
  1- 0 DO
  + LOOP  ;
```

Because it is annoying to count the terms "by hand," suppose we define ((and)) delimiters, leading to:

```
VARIABLE STACKDEPTH
: (( DEPTH STACKDEPTH ! ;
: )) DEPTH STACKDEPTH @ - ;
```

which allows us to use:

```
(( 7 19 24 -56 113 42 )) ADD
(( 7 9 2 )) ADD
```

Next, suppose we define a routine like ADD for multiplication:

```
: MUL
  1- 0 DO
  * LOOP  ;
```

so that ((7 25 4)) MUL leaves the result of 7 * 25 * 4 on the stack.

Finally, suppose we wish to evaluate nested set operations such as:

```
((
   (( 67 -98 78 )) ADD
   (( 89 98 34 )) ADD
   (( 7485 982 -987 0 )) ADD
)) MUL
```

This does not work. What should be calculated is:

(67 - 98 +78) * (89 + 98 + 34) * (7485 + 987 + 0)

Combine the advantages of stacks and variables...

You might like to enter these problems into your system to challenge yourself. Otherwise, continue reading. Refer to these definitions for a possible solution:

```
: ((
  DEPTH R>
  SWAP >R >R ;

: ))
  DEPTH R> R>
  SWAP >R -  ;
```

With these, can you nest set operations as before and still obtain a correct result? Perhaps you should insert the problem into a definition to avoid entering a "wait forever" loop.

What lessons have we learned? Perhaps that:

- Stacks are complicated to use, but nevertheless effective when used with care.
- Variables are easy to use, but are subject to dangerous misuse in those circumstances when a stack is required.

To provide the advantages of both stacks and variables, I defined a new data declarator, as follows:

```
: SVARIABLE
  VARIABLE 0 ,  ;
```

or:

```
: SVARIABLE
  CREATE 0 , 0 ,  ;
```

SVARIABLE is useful for defining instances of what I call stack variables, i.e., a variable and an associated stack. A stack variable initially consumes two bytes more than a normal variable. With each 16-bit value pushed onto a stack variable, four more bytes of memory will be consumed by the stack variable.

Stack variables can accept the same operations as variables, as the following examples illustrate:

```
SVARIABLE A
5 A !
A ? (prints 5)

7 A !
A ?  (prints 7)
```

Furthermore, stack variables can accept PUSH and POP operations:

```
7 A !
5 A PUSH
9 A PUSH
```

Listing One. Implementing stack variables.

```
: SVARIABLE   CREATE 0 , 0 ,  ; \ initialize to 0 the variable and the pointer

200 CONSTANT MAX-TOTAL-DEPTH      \ Maximum number for the sum of the depths of
VARIABLE FREELIST                 \ the svariable stacks id est the number of
MAX-TOTAL-DEPTH 2*  CELLS ALLOT   \ couples alloted for the free list.
                                  \ The FREELIST variable points to the first
                                  \ POINTER-CELL of the alloted area.
HERE  0 , 0 ,                     \ Leave the limit address on the stack , for
                                  \ use by CELLSLINK and null terminate the list
: CELLSLINK  ( limit-address )
  FREELIST DO  I 2 CELLS +  I !  2 CELLS +LOOP  ;

CELLSLINK  FORGET CELLSLINK       \ Link from freelist to here , then forget.

100 CONSTANT #USERS  \  A trick is used to offer to user variables the
CREATE USER-POINTERS \  possibility to be stack variables.
#USERS CELLS ALLOT   \  Reserve 1 POINTER CELL for any user var.
                     \  Constant #USERS is system dependent FIRST-USER-VAR is
                     \  the  address  of  the user variable with the smallest
                     \  address. LAST-USER-VAR  is  the  address  of the user
                     \  variable with the biggest address.

: INSERT  ( insertendum-addr previous-insertendum-addr -- ) \ Attention
  2DUP  <-<  !                                            ; \ INSERT & EXTRACT
: EXTRACT ( previous-extraendum-addr  --   exracted-addr  ) \ use the address
  DUP @ TUCK                                                \ of the pointer
  DUP 0= ?ERROR" ERROR IN LIST OPERATION"                   \ cell. Not the
  <-<                                                     ; \ data address.

: USER? ( addr -- flag )            \ Is the address between the first and the
  FIRST-USER  LAST-USER RANGE[]  ; \ last user var ? It belongs to a user var?

: D>Paddr ( dataCellAddress  --  PointerCellAddress ) DUP USER?
  IF  FIRST-USER -  USER-POINTERS +   ELSE  CELL +  THEN ;
\ Convert the DATA CELL address to the POINTER CELL address.

: COUPLEGET ( -- pointer-addr-of-free-couple ) \ Request an unnamed svariable
  FREELIST EXTRACT   DUP OFF                   \ from the free list and
  CELL -             DUP OFF                 ; \ initialize it .

: COUPLEGIVE ( couple-data-addr  -- )          \ Give an unnamed svariable
  CELL +  FREELIST INSERT        ;             \ to the free list.

: OLD ( svar-data-addr  -- ) \ Drop the last entry in a stack variable.
  DUP D>Paddr EXTRACT        \ ( svar-data-addr extracted-pointer-addr )
  DUP FREELIST INSERT        \ The extracted couple is linked into the free
                             \ list. Stack as above
  CELL -                     ( svar-data-addr extracted-value-addr )
  <-<                      ; \ transfer value of extracted couple into var

: POP ( svar-data-addr  --  previousValue )  \ Keep old value on stack as
  DUP @ SWAP  OLD ;                          \ the result.

: NEW ( svar-data-addr  -- ) \ works like DUP @ SWAP PUSH
  FREELIST EXTRACT           \ ( svar-data-addr  pointer-addr-of-free )
  DUP CELL -- >R             \ keep data addr of free couple on return stack.
                             \ ( svar-data-addr freeCouplePointerAddr )
  OVER D>Paddr INSERT        \ ( svar-data-addr )
                             \ the free couple gets inserted in the svarlist
  R> >->                   ; \ svar value also in free couple

: PUSH ( value svar-data-addr -- )  \  Push in a stack variable
  DUP NEW !                    ;  \  a value.
```

Listing Two. Stack variable utilities and applications.

```
SVARIABLE STACK-DEPTH

: ((   DEPTH  STACK-DEPTH PUSH    ;    \ Mark stack depth.
: ))   DEPTH  STACK-DEPTH POP -   ;    \ Count pushed items.

: -((  ( n -- )                        \ Mark stack n items before the actual
  >R DEPTH R> -  STACK-DEPTH PUSH ;    \ position.

: GROUP.  ( M[0] M[1] ...M[N-2] M[N-1] N -- )  \ type a "group" of numbers.
  0 DO . SPACE LOOP ;

\ The subsequent word serves to type the contents of the svariable like a
\ stack image with the usuall convention TOS ( = TOP OF STACK ) on the right.

: S? ( svar -- )
  DUP D>Paddr SWAP ( pointer-addr data-addr )
  2 -(( \ mark stack excluding the 2 items : pointer-addr & data-addr
  BEGIN                ( some-values pointer-addr data-addr                   )
    @ SWAP             ( some-values value pointer-addr                       )
    @ ?DUP WHILE       ( some-values value next-pointer-addr                  )
    DUP CELL -         ( some-values value next-pointer-addr next-data-addr )
  REPEAT )) GROUP. ;   \ Count pushed items & type them.

: POP? ( svar -- value true | false )              \ Pop if there is
  DUP D>Paddr @ IF POP TRUE ELSE DROP FALSE THEN ;  \ something to pop.

: VOID ( svar -- )                              \ Void the svariable. Obviously
  BEGIN  DUP POP? WHILE DROP  REPEAT  DROP ;  \ one value is always there.

: BASE<   BASE NEW   ;   \  These two words, are useafull if you want to
: BASE>   BASE OLD   ;   \  change base, saving the old one and restoring
                         \  it later. They can be used like:
                         \
                         \    BASE< HEX
                         \    ...........................
                         \    ...........................
                         \    BASE>
```

```
A ?         (prints 9)
16 A !
A ?         (prints 16)
A POP  .    (prints 16)
A ?         (prints 5)
A POP  .    (prints 5)
A ?         (prints 7)
A POP       (error)
```

There are many ways to take advantage of the dual nature of stack variables. For example, to solve the earlier problem of evaluating nested set operations, try:

```
VARIABLE STACK-DEPTH
: (( DEPTH STACK-DEPTH PUSH ;
: )) DEPTH STACK-DEPTH POP - ;
```

With one additional stack operation, `OLD`, a shorter and clearer way to recall a previous `BASE` setting is possible. `OLD` throws away the last entry in a stack variable.

```
: OLD ( svar -- )
  POP DROP  ;
```

```
: BINARY.  ( n -- )
  2 BASE PUSH
  . BASE OLD ;
```

Compare this with the alternative definition of `BINARY` that does not use stack variables:

```
: BINARY.  ( n -- )
  BASE @ SWAP ( keep old base)
  2 BASE ! ( change to binary)
  . BASE !  ;
  ( print, return to old base)
```

Suppose we want to define a word that prints the time on the top right of the screen without affecting the original cursor position (perhaps as part of an interrupt-driven routine). Using stack variables and `PUSH`, `POP`, and `OLD`, you get this:

```
: TIME.
  60 CURSORX PUSH
  0 CURSORY PUSH
    TIMEGET TIME>$ TYPE
    CURSORX OLD
    CURSORY OLD ;
```

This is again clearer and shorter *[and much less prone to stack errors]* than the definition that does not use stack variables:

```
: TIME.
  CURSORX @   CURSORY @
  60 CURSORX !    0 CURSORY!
    TIMEGET TIME>$ TYPE
    CURSORY ! CURSORX ! ;
```

These examples show how definitions can frequently be made more self-documenting and shorter through the use of stack variables. (Because there would be fewer calls to NEXT, these definitions are also faster in cases where PUSH, POP and OLD are coded as primitives.)

It is also easier to create the new definitions. For example, a drawing routine may need to store the current color, switch to a new color, draw, then restore the old color (turtle position and other parameters, perhaps, as well). Such problems arise frequently and all are much more easily solved through the use of stack variables.

Stack variables can even replace local variables, although some additional alterations are needed to provide a clean implementation. Of course, recursion is supported by virtue of PUSH and POP operations. If you develop something promising in this regard, please let me know about it. *[These stack variables are really global in scope, and memory can only be released insofar as POP operations release two cells back into a pool of memory that is shared by all stack variables.]*

Before offering some of the implementation details, allow me to share several more examples of the usefulness of stack variables. Listing Three shows how assembler control structures can be implemented using stack variables. Listing Four shows how locals could be defined. This implementation of locals is notable primarily because of its brevity, not because of its elegance.

Finally, I offer in Figure One a helpful development aid that uses stack variables. INCLUDE accepts a string address which holds the filename of the file to be loaded (obviously, my Forth supports DOS files). INCLUDE requests can be safely nested, because interpretation will reliably continue at the correct resumption points regardless of what the system security and compiler words do with the parameter stack.

Implementing Stack Variables

Stack variables are based upon lists. Lists are formed as a series of list members.

Listing Three. Flow-of-control-definitions.

```
\ BRANCH-CONDITIONALY-BACKWORDS (  condition  address-where-to-jump  )
\ The word compiles a backword jump with the right offset or address.
\ Obviously system dependent definition.
\ BRANCH-CONDITIONALY-FORWARD  ( condition -- )
\ The word compiles a forward jump with the given condition
\ and allots the space for the actually unknown offset or address.
\ The space for the offset ( or address ) will be filled by OFFSET! .
\ OFFSET! ( reference-address destination-addr-of-jump )
\ The above word , given the destination address, and the address
\ of the already compiled jump ( or more precisely :the address immediatelly
\ after the compiled jump ) fills the space left by
\ BRANCH-CONDITIONALY-FORWARD with the correct offset or address.

SVARIABLE CONTROL-BEGINING  SVARIABLE CONTROL-LIST
\ Control structures ( CS ) have a begining and an end.
\ Begining is marked with CONTROL<, while end is marked by CONTROL>,
\ Words between CONTROL<, and CONTROL>, can jump conditionally to the
\ begining or to the end of the structure.Jumps to the begining are
\ immediately resolved while forward jumps will get resolved by CONTROL>,
\ The variable CONTROL-LIST contains an unnamed svariable that contains the
\ addresses of the unresolved forward jumps.
\ The variable CONTROL-BEGINING contains the address of the begining of the CS
: CONTROL<,   HERE CONTROL-BEGINING PUSH   COUPLEGET CONTROL-LIST PUSH ;
: FORWARDRESOLVE ( svar-containing-unresolved-jumps  --   ) locals| V |
  BEGIN V POP? WHILE ( reference-address ) HERE OFFSET! REPEAT ;
\ resolve forward jumps until no more addresses are available.
: CONTROL>,    CONTROL-LIST @ FORWARDRESOLVE  \ At CONTROL>, ends the control
               CONTROL-LIST POP COUPLEGIVE    \ structure. Resolve the pending
               CONTROL-BEGINING OLD           \ forward jumps and go back to
;                                             \ the outer CS.
: WHEN, ( condition   --   )
  BRANCH-CONDITIONALY-FORWARD   HERE  CONTROLLIST @ PUSH ;
\ jump to end of control structure ( CS ) if the condition is true
: WHILE, ( condition  --   )   CONDITION-NEGATE WHEN, ;
\ jump to end of CS if condition is false
: LEAVE,  TRUE, WHEN, ; \ jump to end of CS always
: ELSE,   \ A forward jump to the end is compiled.
          \ Branches to end are resolved and redirected to the portion
          \ of code after ELSE, that represents the faillure code.
  TRUE,  BRANCH-CONDITIONALY-FORWARD
  CONTROLLIST @ FORWARDRESOLVE
  HERE CONTROLLIST @ PUSH  ;
\ ELSE may be used in any control structure , not only in IF, THEN,
: ?AGAIN,( condition  --   )
  CONTROL-BEGINING @ BRANCH-CONDITIONALY-BACKWORDS ;
\ jump to begining of CS if condition is true
: AGAIN,   TRUE, ?AGAIN, ; \ jump always to the begining of CS
: IF,    ( condition  --   ) CONTROL<, WHILE, ;
: THEN,     CONTROL>,    ;

: REPEAT<,   CONTROL<, ;
: REPEAT>,   AGAIN, CONTROL>, ;
: UNTIL>,    ( condition  --   ) CONDITION-NEGATE ?AGAIN, CONTROL>, ;

\ With that words any number of whiles may exist in one control-structure
\ andif's are simple:  : ANDIF, ( condition -- ) WHILE, ;
\ " cond IF, ...cond ANDIF, ... cond ANDIF, success-code THEN, " or
\ " cond IF, ...cond ANDIF, cond ANDIF, success-code ELSE, failure-code THEN,"
\ obviously it isn't necessary to define ANDIF,  WHILE, does the work:
\ " cond IF, ...cond WHILE, ... cond WHILE, success-code THEN, " ...
\ Other examples are:
\ REPEAT<, ... cond WHILE, ... cond WHEN, ... REPEAT>,  ecc.
```

Listing Four. Local variables, old and new.

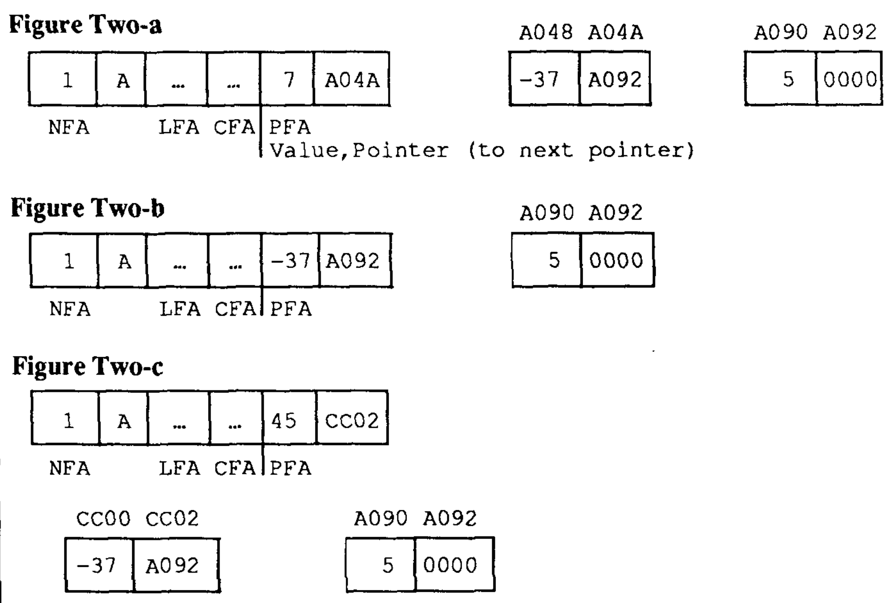
```
: DROPS ( n -- ) 0 DO DROP LOOP ; \ Better if in machine language.

: CALLERS-SWAP       \ The execution of the calling and the pre-calling word
  R> R> SWAP >R >R   \ are inverted. First is concluded the precalling word and
;                    \ after the calling. ( Orrible unstandard trick ) .

\ Local variables must be defined as svariables before beeing used.
\ Only one dictionary entry is needed for locals with the same name in
\ different words.
\ The words  "((" and  ")) Loc " are used preferably at the begining of a
\ definition . EXAMPLE:
  : ELIPSE ( x y a b ) \ the word elipse gets 4 numbers on the stack
    (( X Y A B )) LOC  \ the 4 numbers are respectively pushed into the
    X @ Y @  MOVETO    \ 4 stack-variables X Y A B. After completion of the
    A @ B @  CIRCLE    \ word the 4 variables will be "OLDed".
  ;
\ The svariable LOCALS-LIST is used to keep unnamed svariables ( couples )
\ that keep the svariables that must be olded at the end of the definition.
0 CONSTANT N   0 CONSTANT V  \ used as "TO VARIABLES"
SVARIABLE LOCALS-LIST
: Loc ( svar[N-1] svar[N-2] ... svar[0] N -- )
  [ ] N  !   COUPLEGET  ['] V  !
  N 0 DO  DUP V PUSH  N PICK SWAP PUSH LOOP  N DROPS   V LOCALS-LIST PUSH
  CALLERS-SWAP
  \ at the end of the word containing the LOC execution will continue here
  LOCALS-LIST POP  ['] V  !
  BEGIN  V POP?  WHILE OLD REPEAT  V COUPLEGIVE
;
\ we could define   : (  (( ;  and   : }   )) LOC ;  ( but attention ... )
\ So we could do   : QUADRATIC { a b c x }
\                    a @ x @ * x @ *  b @ x @ * + c @ + ;
```

```
: ->  ( value addr  -- ) ! ;       \ For completness , never used.
: <-  ( addr value  -- ) SWAP ! ; \ Faster then SWAP ! , if primitive

: <-< ( DestAddr   SourceAddr -- ) @ <- ;      \
: >-> ( SourceAddr DestAddr   -- ) SWAP <-< ;  \
\ The above two words serve to copy the value of a variable into
\ another variable . They are used like " A B >-> "  or " B A <-< "
```

Figure One. A nestable INCLUDE.

```
. [illegible]DE   ( filename$ -- )
  $>FILE INFILE PUSH  ( opens files, & pushes )
                      ( file pointer to var. INFILE )
\ The symbol % means chunk: a piece of memory,
\   somethina like a counted strina:

      LE @ FILE>%
\ Allocates memory & copies file contents there.
\ The chunk pointer is left on the stack.

  INTERPRETED% PUSH
\ Pushes chunk pointer to the SVARIABLE containing
\   the actual chuink being interepreted.

  0 >IN PUSH
\ Saves old >IN and begins interepreting new chunk.

  INTERPRET
  INFILE POP FILECLOSE
  INTERPRETED% POP FREE%
\ Closes file & frees memory allocated for file.

  >IN OLD  ;
\ Restores the old >IN.
```

Figure Two. Cell couples as list elements.



Each member is comprised of two adjacent memory cells that I will call a cell couple, or couple. By cell, I mean two consecutive bytes in 16-bit systems, or four consecutive bytes in 32-bit systems.

Of the two cells in a couple, one holds the arbitrary data value. The other cell holds a pointer, the address of the next member in the list. It points at the pointer cell of the next couple. The final member of a list always contains zero within the pointer cell (you may change to a new list-terminator value, but it must be used consistently).

To show how data values are stored in a stack variable, suppose you enter the following instructions:

```
SVARIABLE A
    5 A !
  -37 A PUSH
    7 A PUSH
```

In memory, the result will be as shown in Figure Two-a. After entering `A POP .` memory is updated as in Two-b. The cell couple formerly at A048 was returned (linked) to the free list.

Suppose the next thing you enter is `45 A PUSH`. This requires the removal (de-linking) of a cell couple from the free list for use to store the new value. If the allocated cell couple is at CC00, then memory would be updated as shown in Figure Two-c.

To summarize the requirements of stack variable implementations, we need to be able to:

- Reserve space for the free list couples.
- Allocate a couple by de-linking it from the free list and linking it to the stack variable.
- Deallocate a couple by de-linking from a stack variable and linking it to the free list.

Two primitives are used to manage couple lists, INSERT and EXTRACT. Their actions are shown in the "before" and "after" illustrations in Figures Three and Four.

The <- word is simpler to remember than a `SWAP !` phrase, particularly for beginners who are more likely to enter an (often fatal) sequence like:

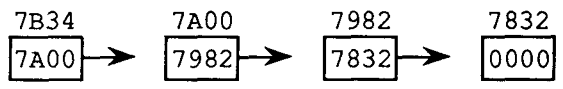
```
A 5 !
```

For movement of data between two variables, a pair of primitive operations has

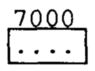
Figure Three. The definition and actions of `INSERT`.

```
: INSERT   ( insertendum-adr prev-insertendum-adr -- )
  2DUP  <-< !  ;
```

Below, only the pointer cells are illustrated. If we start with this:



while cell 7000 is available,
and we perform
`7000 7A00 INSERT`
the result will be:



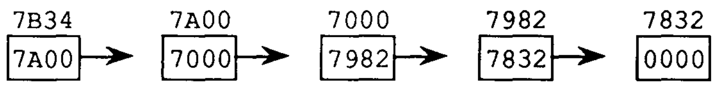
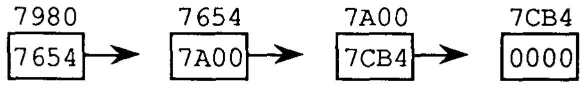


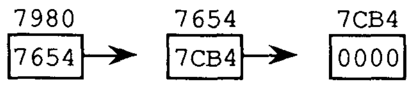
Figure Four. The definition and actions of `EXTRACT`.

```
: EXTRACT   (prev-extraendum-adr -- extracted adr )
            (DUP @ TUCK <-< )
  DUP @ TUCK DUP 0=
  ?ERROR" Error in list operation"  <-<  ;
```

If we start with:



and then perform `7654 EXTRACT` we will have
7A00 on the stack and, in memory:



been defined similar to <- and ->, namely <-< and >->. *[Do not confuse any of these with the prefix* `TO` *operation of some Forths—this is purely postfix.]*

The code for implementing stack variables is provided in Listing One. Listing Two provides definitions for a few supplementary words that are handy for dealing with stack variables. It also exhibits a few more tiny applications for stack variables.

Both listings reveal how simple stack variables really are. But like any new extension of Forth, many more possibilities are brought to light, both syntactical and conceptual. I hope you will enjoy exploring more of the possibilities.

Conclusion

The stack variable concept isn't applicable only to Forth. In fact, I felt the need for something like it while working in Turbo Pascal with a graphics package (trying to decide whether the user must restore his drawing color upon return from a called procedure, or whether the called routine must save and restore the color). If you know how to implement stack variables in Pascal, C, or BASIC coherently with the rest of the language, you are probably a compiler producer. I am not, but I try to be a Forth programmer.

Hopefully, Forth gives us the freedom not only to "feel the need," but also to "resolve the problem."

(Continued from page 4.)

variables and routines. This is beyond the provisions of most macro facilities, but is not as efficient and, at the moment, cannot be extended to include immediate words (the compiler is extended, but not in the usual ways).

"The implementation does not involve dynamic memory management. Local variables are alluded to, but this is not concerned with meeting all the requirements locals must satisfy.

"'Stack Variables' by Giorgio Kourtis. His new data definer creates instances of stacks via a technique that can replace local variables altogether. But no way is given here to deallocate a stack variable, and it is really a globally accessible object. There are many points to note in this sophisticated piece; one is how multiple stack objects share statically allocated memory with the greatest of implementation ease.

"The implications could be far ranging. The author shows how many Forth routines can be reimplemented with stack variables, which could make Forth more robust and even more flexible. *[Any implementors care to help confirm or deny this?]* This might be the foundation for implementing functional languages in Forth (APL, anyone?). An implementation of locals is shown, naturally, but complex stack gyrations accompany this approach just as much as they accompany the traditional ones."

BANKING ON THE R65F11

D.C. EDWARDS - PERTH, W.A., AUSTRALIA

A bank of memory is a subsection of a processor's memory, in which many different physical memory chips can be selected separately, each being referred to as a bank. The selection of the memories is performed by hardware (usually an output port), which works in conjunction with the address decoding logic to produce the different bank memory device select lines.

The bank selection hardware is set, of course, by software instructions, so bank systems involve much more interaction between hardware and software than do other architectures. During development, the hardware designer must work to simplify and unify the bank interface software, and the software designer must have a detailed understanding of the hardware's response to the software commands.

Bank memories have been used in computer architectures for various reasons: to extend the memory (and/or types of memory) in the system, for communications (where they act as distinct buffers), and in multiprocessing environments (where they allow simultaneous data transfers between sets of processors). Bank memories offer the multiprocessor system designer the speed and reliability of closely coupled architectures, while retaining the processing independence of loosely coupled architectures.

One of the major problems with bank architectures is that bank operating systems are usually inefficient—one writer has said that they are "destined to be a kluge." This is because such operating systems usually either overload the system bank (by forcing it to handle all administration of calls to other banks), or they copy system information into all banks (to distribute administration), ironically making them memory inefficient.

Forth is a language famous for hardware interfacing and for its power as a consistent operating system, and thus presented itself as the language in which to construct an entire Bank Operating System (the BOS!).

Such an operating system is described in this paper, and was developed on the Rockwell R65F11, one of the first "Forth on board" processors released (July 1983). It has a modest external memory space of 16,128 bytes, which is small even for Forth developers. This limited memory was the main impetus to develop the BOS: in a single-bank F11 development system, 9K is consumed by the Forth system, leaving 7K for applications. Consequently, target machines developed on such a system are also limited to 7K. A bank version of the F11 development system allows development of target systems which could utilize the full 16K of an F11 bank, as well as allowing multiple-bank target systems.

Bank operating systems are usually inefficient.

After completing development systems which use up to seven banks (112K) and target machines which use four banks (64K), I can report that Forth does indeed deliver a transparent and efficient operating system simply by developing a modified `INTERPRET` loop.

F11 Resources

The memory space of the F11 is segmented into three sections:

on-chip ROM ($FCFF–$FFFF)
Holds the Forth kernel code.

on-chip RAM ($0040–$00FF)
Used for the Forth stacks.

external memory ($0100–$3FFF)
Available for multiple bank memories.

We shall see later that it is actually necessary to have such a segmented memory space to implement a bank system.

The F11 uses an on-chip port (port B) to output an eight-bit bank address, providing up to 256 banks. The lines of port B are thus connected into the hardware decoding logic to perform the bank memory selection. All of the banks appear to be in the external memory locations ($0100–$3FFF). The F11 boots up with port B at $FF, so bank FF is the boot bank in which the system code must reside. Port B can be thought of as providing eight address lines (A14–A21) in addition to the F11's A0–A13.

The Forth on the F11, RSC-FORTH, follows the Forth-79 Standard, with extra words to access the F11's environment, and includes three bank memory interface words. These words use arguments similar to the standard Forth words after which they are named, with an extra stack entry to specify the memory bank. The words are:

`BANKC@` (A b - - d)
D is byte-fetched from address A in bank b.

`BANKC!` (d A b - -)
Byte d is stored into address A in bank b.

`BANKEXECUTE` (A b - -)
Word at address A in bank b is executed.

All of these words have the same form:

```
1. Save current bank      (PB C@ >R      )
2. Set up new bank        (PB C!         )
3. Operation    (C   @ C! EXECUTE )
4. Restore old bank       (R> PB C!  )
```

Port B is saved on the return stack, so the operations can be nested to any practical depth. When one of these bank words is run, a new bank of external memory is selected (by the hardware connected to port B) during step two. This means that all of the memory contents between $0100 and $3FFF have changed to the values in the new bank. At step four, when port B is restored, the external memory (between $0100 and $3FFF) reverts to its previous values.

This fact, that the decoding hardware selects a different physical memory during the bank operation, is the hardware process of which the software writer must have the most detailed understanding, and it has the following important consequences for the system.

1. Bank primitives must be on chip.

If these primitive bank programs were located in external memory, the processor's program counter would be pointing to external memory during the bank operation. When the new bank is selected, the program counter would be at some address within the new bank (most likely not pointing to well-formed op-codes), causing disaster. When located in ROM (or Zero Page), the program counter is always pointing to on-chip addresses, ensuring that the entire operation is safely managed. This is the reason that a bank architecture must have a segmented memory space: the decoding must ensure that some section of the memory is common to all banks, in which the bank primitives can reside (hardware designers note!).

2. On-chip kernel words may be bank executed.

We can exploit the fact that a different bank is selected during BANKEXECUTE to provide BANK versions of all the kernel processes. To do this, instead of simply executing the address of a kernel process, we push a bank value on top of the address and BANKEXECUTE the composite bank+address. This process runs the kernel process while the named bank memory is being selected. If the kernel word references external memory, the operation will be performed on the selected bank memory and not on the memory of the calling bank. For example, the phrase:

```
ADDRESS '
@ CFA  FC
BANKEXECUTE
```

will fetch from the ADDRESS within bank FC. Such bank execution allows us to define words like those below.

Fetch word n from address A bank b:

```
: BANK@ ( A b -- n )
  [ ' @ CFA ] LITERAL SWAP
  BANKEXECUTE ;
```

Store word n to address A bank b:

```
: BANK!   ( n A b -- )
  [ ' ! CFA ] LITERAL SWAP
  BANKEXECUTE ;
```

Add n to value in address A bank b:

```
: BANK+! ( n A b -- )
  [ ' +! CFA ] LITERAL SWAP
  BANKEXECUTE ;
```

This process can be extended beyond fetch and store words to, for example, the dictionary search word (FIND):

```
: BANKFIND
( 'NAME 'Dict B --Addr Bnk Cnt flag )
  [ ' (FIND) CFA ] LITERAL SWAP
  BANKEXECUTE ;
```

During BANKFIND, CURRENT, the name strings, and links are all fetched from the named bank, not from the calling bank. Like the bank primitives, these words need a bank number to be pushed onto the stack before invocation.

3. User variables in each bank.

As the USER variables are located in the external memory, each bank has its own private copies of the system USER variables. This allows, for instance, different banks to work with different bases. Compiling code independently in each bank is possible, as each one has private copies of the variables used to administer compilation: DP, CURRENT, etc.

We can, thus, define a set of bank words similar to Forth's.

Current dictionary pointer in bank b:

```
: BANKHERE ( b -- n )
  DP SWAP BANK@ ;
```

Increment bank b dictionary pointer by n:

```
: BANKALLOT ( n b -- )
  DP SWAP BANK+! ;
```

Again, these words require the bank number on the stack before execution.

Bank Operating System

The design aims of Jarrah Computers' Bank Operating System were:

- Allow compilation of code in banks other than the bootup bank ($FF).
- Allow headers to be built in a bank separate from the code.
- Remove the need for bank-number statements in the text (leaving the text as close to standard Forth as possible).
- Allow words to call each other as in standard Forth (irrespective of their definition bank and the current compile bank).

The Compile Bank

To remove the need for explicit bank declaration, the variable ,BANK was created to hold the value of the compile bank, which is the default bank for all bank operations. The value in ,BANK (together with standard 16-bit addresses) forms a 24-bit bank+address used in most of the bank operations. To facilitate this, we define (usually in code):

```
: BANK ( -- n )
  ,BANK @ ;
```

(pushes ,BANK value onto stack)

Combining this with the BANK words, we can define words of the form:

```
: BC@ ( A -- n )
  BANK BANKC@ ;

: B!  ( n A -- )
  BANK BANK!  ;

: B+! ( n A -- )
  BANK BANK+! ;
```

These words now accept the same arguments as standard Forth, so "Bname" words, as distinct from BANKname words, use the same arguments as Forth.

Compiling Words Defined in Remote Banks

When we are compiling in a bank and encounter a word defined in some other bank, we compile the call to the remote word using the following code structure:

```
Addr ——→ BEXEC
Addr +cell ——→ Bank
Addr+2cells——→ Addr
```

The word BEXEC fetches the remote bank and address from Forth's code stream and then bank executes it:

```
: BEXEC ( -- )
\ runs bank+address next to
\ itself in code stream)
  R> DUP  4 + >R  2@
  BANKEXECUTE ;
```

The code structure is built by BANK, (e.g., BANK COMPILE). It expects an address and the bank-of-compilation of a word on the stack, and it tests to see if the word is defined in the current BANK or in the KERNEL. If so, BANK, compiles the address, as in standard Forth compilation; otherwise, BANK, compiles the BEXEC followed by the bank, and finally the address.

```
: BANK, ( A CompileB -- )
  OVER F400 U< NOT
  OVER BANK = OR
  IF DROP
  (the bank number)
  ELSE COMPILE BEXEC ,
  (compile BEXEC and Bank)
  THEN , ;
  (compile the address)
```

Whilst this method consumes three times the space of a standard compilation, it leaves a completely transparent outer loop—any other system would burden the interpreter with the task of determining how to run what word from where. A Forth compiler was then written using these default bank operations. For example:

```
: HERE
  DP B@ ;

: ALLOT
  DP B+! ;

: ,
  HERE B! 2 ALLOT ;
```

Headers

To compile the headers, a variable HBANK was created to hold the header bank. Compilation in the header bank uses a word H[which swaps the values in the ,BANK and the HBANK:

```
: H[
  ,BANK @ HBANK @
  ,BANK ! HBANK ! ;
```

After reading cmFORTH, I decided to define an alias of H[called]H so that syntaxes of the sort H[*...Forth statements ...*]H could be written, and all statements between H[and]H will be executed/compiled from/to the HBANK. For example, the phrase n H[,]H will compile n into the header bank and increment the header bank's DP, etc. Note that H[n ,]H works identically.

With this utility, the header construction words (LATEST, CREATE, etc.) were defined using the bank words and H[*...bank.words...*]H syntaxes. For example:

```
: LATEST
  CURRENT B@ B@ ;

: SMUDGE
  H[
    LATEST DUP
    BC@ 20 XOR
    SWAP BC!
  ]H ;
```

Header Structure

The structure of a header is dependent on the current compile and header banks. If they are equal, a standard RSC-FORTH header is created. If the compile bank is not equal to the header bank, the header structure is different.

Standard, or local, header:
```
|Cnt|NAME|Link|PFAPTR|
```

Bank, or far, header:
```
|Cnt|NAME|Link|Bank|PFAPTR|
```

FIND returns the same arguments as -FIND in RSC-FORTH, with an extra number indicating the bank in which the header was located.

Forth's CFA had to be extended to fetch both the CFA *and* the compile bank from the data returned by FIND. It was named CFAR, and uses the different header structures to determine the compile bank:

THINK BIG

MMSFORTH

THIRTY-DAY FREE OFFER — Free MMSFORTH GAMES DISK worth $39.95, with purchase of MMSFORTH System. CRYPTOQUOTE HELPER, OTHELLO, BREAKFORTH and others.

[illegible]

HARVARD SOFTWORKS

NUMBER ONE IN FORTH INNOVATION

(513) 748-0390 P.O. Box 69, Springboro, OH 45066

MEET THAT DEADLINE ! ! !

- Use subroutine libraries written for other languages! More efficiently!
- Combine raw power of extensible languages with convenience of carefully implemented functions!
- Yes, it is faster than optimized C!
- Compile 40,000 lines per minute!
- Stay totally interactive, even while compiling!
- Program at any level of abstraction from machine code thru application specific language with equal ease and efficiency!
- Alter routines without recompiling!
- Use source code for 2500 functions!
- Use data structures, control structures, and interface protocols from any other language!
- Implement borrowed feature, often more efficiently than in the source!
- Use an architecture that supports small programs or full megabyte ones with a single version!
- Forget chaotic syntax requirements!
- Outperform good programmers stuck using conventional languages! (But only until they also switch.)

HS/FORTH with FOOPS - The only flexible full multiple inheritance object oriented language under MSDOS!

Seeing is believing, OOL's really are incredible at simplifying important parts of any significant program. So naturally the theoreticians drive the idea into the ground trying to bend all tasks to their noble mold. Add on OOL's provide a better solution, but only Forth allows the add on to blend in as an integral part of the language and only HS/FORTH provides true multiple inheritance & membership.

Lets define classes BODY, ARM, and ROBOT, with methods MOVE and RAISE. The ROBOT class inherits:

```
  INHERIT> BODY
  HAS> ARM RightArm
  HAS> ARM LeftArm
```

If Simon, Alvin, and Theodore are robots we could control them with:

```
Alvin 's RightArm RAISE       or:
+5 -10 Simon MOVE             or:
+5 +20 FOR-ALL ROBOT MOVE
```

Now that is a null learning curve!

WAKE UP ! ! !

Forth is no longer a language that tempts programmers with "great expectations", then frustrates them with the need to reinvent simple tools expected in any commercial language.

HS/FORTH Meets Your Needs!

Don't judge Forth by public domain products or ones from vendors primarily interested in consulting - they profit from not providing needed tools! Public domain versions are cheap - if your time is worthless. Useful in learning Forth's basics, they fail to show its true potential. Not to mention being s-l-o-w.

We don't shortchange you with promises. We provide implemented functions to help you complete your application quickly. And we ask you not to shortchange us by trying to save a few bucks using inadequate public domain or pirate versions. We worked hard coming up with the ideas that you now see sprouting up in other Forths. We won't throw in the towel, but the drain on resources delays the introduction of even better tools. Don't kid yourself, you are not just another drop in the bucket, your personal decision really does matter. In return, we'll provide you with the best tools money can buy.

The only limit with Forth is your own imagination!

You can't add extensibility to fossilized compilers. You are at the mercy of that language's vendor. You can easily add features from other languages to **HS/FORTH**. And using our automatic optimizer or learning a very little bit of assembly language makes your addition zip along as well as in the parent language.

Speaking of assembly language, learning it in a supportive Forth environment turns the learning curve into a light speed escalator. People who failed previous attempts to use assembly language, conquer it in a few hours or days using HS/FORTH.

HS/FORTH runs under MSDOS or PCDOS, or from ROM. Each level includes **all** features of lower ones. Level upgrades: $25. plus price difference between levels. Sources code is in ordinary ASCII text files.

All HS/FORTH systems support full megabyte or larger programs & data, and run faster than any 64k limited ones even without automatic optimization -- which accepts almost anything and accelerates to near assembly language speed. Optimizer, assembler, and tools can load transiently. Resize segments, redefine words, eliminate headers without recompiling. Compile 79 and 83 Standard plus F83 programs.

STUDENT LEVEL $145.
text & scaled/clipped graphics in bit blit windows,mono,cga,ega,vga, fast ellipses, splines, bezier curves, arcs, fills, turtles; powerful parsing, formatting, file and device I/O; shells; interrupt handlers; call high level Forth from interrupts; single step trace, decompiler; music; compile 40,000 lines per minute, stacks; file search paths; formats into strings.

PERSONAL LEVEL $245.
software floating point, trig, transcendental, 18 digit integer & scaled integer math; vars: A B * IS C compiles to 4 words, 1..4 dimension var arrays; automatic optimizer-machine code speed.

PROFESSIONAL LEVEL $395.
hardware floating point - data structures for all data types from simple thru complex 4D var arrays - operations complete thru complex hyperbolics; turnkey, seal; interactive dynamic linker for foreign subroutine libraries; round robin & interrupt driven multitaskers; dynamic string manager; file blocks, sector mapped blocks; x86&7 assemblers.

PRODUCTION LEVEL $495.
Metacompiler: DOS/ROM/direct/indirect; threaded systems start at 200 bytes, Forth cores at 2 kbytes; C data structures & struct+ compiler; TurboWindow-C MetaGraphics library, 200 graphic/window functions, PostScript style line attributes & fonts, viewports.

PROFESSIONAL and PRODUCTION LEVEL EXTENSIONS:

FOOPS+ with multiple inheritance**$ 75.**
286FORTH or 386FORTH $295.
16 Megabyte physical address space or gigabyte virtual for programs and data; DOS & BIOS fully and freely available; 32 bit address/operand range with 386.
BTRIEVE for HS/FORTH (Novell) **$199.**
ROMULUS HS/FORTH from ROM**$ 95.**
FFORTRAN translator/mathpak **$ 75.**
Compile Fortran subroutines! Formulas, logic, do loops, arrays; matrix math, FFT, linear equations, random numbers.

```
: CFAR  ( PFA HB --CFA BANK )
\ Gets BANK & Run addr from Hdr
  2DUP BANK@ DUP >R
  100 U< IF >R   2+
  R> BANK@ 2- R>
       ELSE SWAP DROP R>
       2- SWAP
       THEN ;
```

The BOS Interpreter

We now have all the utilities we need to construct the bank interpreter. The process is formally the same as a standard Forth interpreter, with extensions to handle the bank aspects of the code structure. After `BEGIN WORD` ... we have something like:

```
FIND IF >R CFAR R>
  STATE @ < IF BANK,
    ELSE BANKEXECUTE
  THEN
ELSE NUMBER ...
```

After `FIND` we have the header bank on top of the stack, used by `CFAR` to fetch the compiled bank.

If compiling, then `BANK,` is run to compile the word. If executing, `BANKEXECUTE` runs the word. It's that simple.

In conclusion, I can happily say that all of the design criteria were met. The bank text is loadable by other Forth-79s, with the exception of `H[ ... ]H`, which can be quickly edited out of the way when using bank code in other environments. The run-time speed of the machine is hardly impaired, as most of the work is done at compile time. The BOS provides the 6502-based F11 with:

- a transparent Forth with access to four Mbytes of RAM.
- a 16-bit standard Forth and/or a 24-bit 'bank' Forth.
- Two levels of syntax—`BANK`words and `B`words.

After completing this project, I was struck by the similarity between bank memories and Forth itself. Banks are like Forth for memories—they are modular units, can be extended at will, and—through the BOS—are transparently accessed.

Without Forth's modularity, compiler extensibility, and "conveniences" like `USER` variables per bank, etc., this program would have been difficult, if not impossible, to write.

Dave Edwards founded Jarrah Computers, a company specializing in the design of custom microcontrollers, from the 68705 single-chip family to large industrial systems based on Rockwell's 65F11 Forth chip and, recently, Motorola's 68HC11.

UPPER DECK FORTH $49

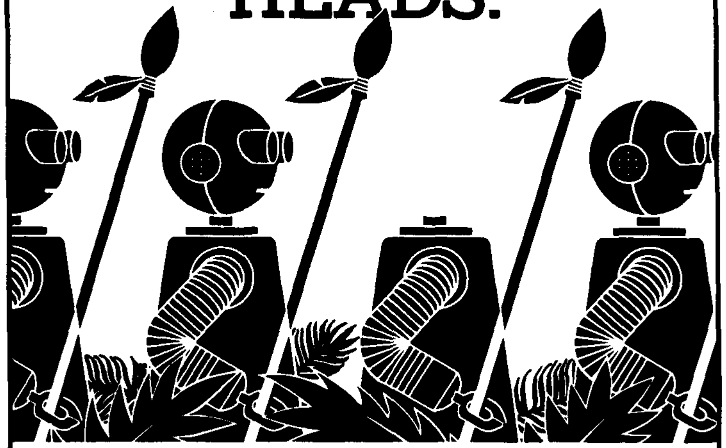
- Based on Forth-83 Standard
- Fully segmented architecture
- Uses ordinary ASCII text files
- Direct threaded code with top of stack in register for fast execution
- Compiles 32K file in 6 seconds on 4.77 MHz IBM PC
- Built-in multi-file full screen editor
- Assembler, decompiler, source-level debugger
- Turnkey application support, no royalties
- Complete documentation
- For IBM PC/XT/AT and compatibles with 256K, hard disk or floppy, DOS 2.0 or later

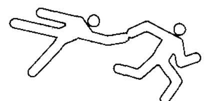
Add $3 for shipping and handling (outside USA $15). CA residents add sales tax.

UPPER DECK SYSTEMS

P.O. Box 263342, Escondido, CA 92026
(619) 741-1075

WE'RE LOOKING FOR A FEW GOOD HEADS.





DASH, FIND
ASSOCIATES
Forth Recruiters

70 Elmwood Ave./Rochester, NY 14611/(716) 235-0168

BEST OF GENIE

GARY SMITH - LITTLE ROCK, ARKANSAS

News from the GEnie Forth RoundTable—Object-oriented programming is a hot topic once again. This is not only true in the computer world as a whole, but also in the community of Forth users.

Object-oriented programming, or OOP, first entered computer parlance with the creation of Smalltalk at Palo Alto Research Center (PARC), though at the time few people really understood object-based languages. In essence, an object-based language organizes knowledge by groups, or objects. In more traditional languages these would be records, or perhaps lists. Objects can exchange data (referred to as 'sending messages') on which methods (i.e., procedures) can be performed. Objects are permitted to share methods.

If OOP and OOF are still a mystery to you, it is clear you have not been following the Object-oriented Forth message topic on the GEnie Forth RoundTable. The following excerpts should make you wish you were participating. I can promise your comments on this and dozens of other interesting topics are more than welcome.

* * *

Topic 41
Fri Nov 03, 1989 OLORIN [Dave]
Sub: Object-Oriented Forth

(For the discussion of object-oriented Forth systems, including the extension to F-PC that can be found in the Forth RoundTable library.)

H.SABBAGH [hadil]

Dave: Downloaded and looked over your Object-Oriented Forth extensions yesterday. I am very impressed: it is some of the *best* Forth code I have read in a *long* time. I am trying to implement this stuff on HS/Forth. I don't quite understand the definitions found in the INTERNALS.SEQ file, since I don't own F-PC. Clearly, `forward:` and `define:` are used for some kind of deferred word. Could you shed some light on this? I am not a rank beginner, so I'm really after data on the functionality of these words; I am sure that I'll have to completely reimplement them for HS/Forth, since the vocabulary is *much* different.

Forth controls binding through various mechanisms.

Extensions that I would like to work on:

- Port to HS/Forth.
- Add variable-length objects. The class is defined to support variable-length objects; instantiating a class requires a count of the number of cells needed. This is available in Smalltalk also, but is non-trivial to build.
- Add dynamic allocation. Since objects are referred to by their address, this shouldn't be too hard to integrate with the rest of the code. It will also add the ability to use local variables, arrays, and stuff like that. I would use something like a C++ memory model, where objects clean up after themselves, although there are theoretical advantages to garbage collection.
- Experiment with persistent object, i.e., object-oriented, databases.

One can slightly modify your code to add an extra level of indirection, i.e., object-pointers are not memory addresses, but indexes in some *big* table. This table would not be in memory at the same time, so I'm talking about some kind of virtual memory system. This idea cannot be implemented as nicely in any other language; in Smalltalk, you need heuristics to discriminate which objects should be handled this way, and in C++ you can't expose object internals easily. With this idea, you write system code in Forth and your database using the OO extensions; the system takes care of the persistence issues transparently.

I hope you will be able to give me advice, etc. Perhaps we can work on this like Dennis' telecom project? I intend to share all source code; I hope it will be as good as yours.

—Hadil

OLORIN [Dave]

Thanks for the comments (they've done my ego no end of good :-)).

Creating dynamic objects isn't that hard... an object structure is a pointer to its class, followed by a block of memory the same size as the size of the object (for the instance variables). With that information, it shouldn't be hard to use it with your memory manager (whatever kind you choose to use).

What `forward:` and `define:` do is allow forward referencing. The word `forward:` creates all of a colon definition, but doesn't have any code in it at all. The `define:` stage allows the code written then to be plugged into the previous define. This is what makes the forward definition as efficient as a regular colon definition.

The internals module needs to be reimplemented for each Forth the package is ported to (by the way, 2.1 is on its way, with a few more words for dealing with classes). The internals module is essentially the abstracted-out dictionary structure words... later on tonight, I'll sit down with the code and write a detailed description of what each word in that module is supposed to do (you *will* need to dig deeply into the dictionary structure and threading mechanism of your Forth to port the code).

Dynamically sized objects: What you might consider doing is having an instance variable called `body^` (a pointer to the dynamically sized portion). By using the ability to define initialization methods (which are executed when an object is created), you can allocate this space at creation time, set the instance variables, and have the rest of the methods work off of the pointer.

(Oh, and any dynamic object creation *does* need to initialize the object.)

—Dave

H.SABBAGH [hadil]

Dave: Thanks. I just finished porting the MODULES.SEQ to HS/Forth; only the word `module` needed to be changed. I will upload this weekend. The dictionary structure is *much* different than that of most public-domain Forths (except a little like BBL). I will have to take some time just to figure out how it works. There has been some mention of creating a separate category. I'm game, but I won't be able to spend more than four to five hours per week on this project. If you all can bear with me, I'll be happy to post my code changes, etc. I am also concerned about HS/Forth vs. other Forths. Let me know what you think. I will try to post a "to-do" list next week.

—hgs

DOJUN YOSHIKAMI
Subj: Smalltalk? C++?

The big difference is Forth was originally made for fast real-time applications where time and space are of essence. Forth is very small, the address interpreter can fit in 1K, a reasonable *extensible* Forth system can be put in 8K of ROM (ROM, mind you). Forth also has control over binding times. From what I have read, OOP languages tend to bind late (Smalltalk is a late binder, apparently) whereas most languages, such as Ada, C, or Fortran, bind early at compile time. Forth has control over binding through various mechanisms (i.e., bind whenever you like) with `CREATE ... DOES>` or deferred words. I've even heard of some guys who wrote a cross-compiler in Forth for the IBM PC because Forth was the only language small and fast enough. Forth is its own operating system, polyFORTH for the VAX comes on ten (?) floppies. (Micro-vms, I think, is about the same size, but that's only the base system, not including utilities, assemblers, and the like.) The only problem is, Forth is so unlike anything else, it's not well known, and since it is for real-time programming, useful data structures such as heaps have to be coded yourself. (Heap management take up heap big time!)

Pardon the pun.

—D. Yoshikami

keith@curry.uchicago.edu
(Keith Waclena)

In response to the following remark: "It seems to me that by the characteristics of Forth, it should be a suitable language for implementing interpreters for languages that are difficult to compile without restricting their operativeness. The languages I'm thinking of are Prolog, Lisp or fully operative object-oriented languages. But the literature doesn't mention such possibility, so I wonder if I'm wrong. Could anybody tell me if any work has been done in this field?"

There has been a lot of this sort of thing in the Forth literature (though it doesn't seem to make it very far into the general C.S. literature). Here are some citations; these citations aren't necessarily recommendations unless explicitly indicated.

L. L. Odette. "Compiling Prolog to Forth," *Journal of Forth Application and Research*, vol. 4, no. 4, pp. 487-533. 1987.

Dick Pountain. "Object-Oriented Extensions to Forth," *Journal of Forth Application and Research*, vol. 3, no. 3, pp. 51-73. 1986. Includes source code. Describes a true object-oriented extension to Forth (including subclass inheritance). [KDW]

Dick Pountain. *Object-Oriented Forth: Implementation of Data Structures*, Academic Press, London. 1987. A more tutorial version of Pountain 1986. Quite good. [KDW]

Christopher J. Matheus. "The Internals of FORPS: A FORth-based Production System," *Journal of Forth Application and Research*, vol. 4, no. 1, pp. 7-27. 1986. Describes an OPS5-like production system for Forth. [KDW]

While I don't have the reference handy, there has been at least one article in the FORML proceedings in the last few years on Lisp in Forth.

These are just a few examples. The two major places to look for articles of this type are the *Journal of Forth Application and Research* and the proceedings of the FORML conferences. Both are refereed. *Forth Dimensions*, the journal of the Forth Interest Group, is another source, although I don't think it's refereed. *[All our articles are subject to technical review, though perhaps in a less formal process than that employed by many academic publishers. —Ed.]*

And regarding this remark, "Just one more question. Could anybody tell me where I can get a public-domain Forth implementation for the 80x86 family?" By mail from the Forth Interest Group (at cost; see order form in this issue), or by anonymous FTP from wsmr-simtel20.army.mil (cd to pd1:<msdos.forth>, set tenex mode, and get either F-PC (a full-featured DTC implementation) or F83 (a simpler ITC implementation, excellent for learning purposes).

—Keith

GENE LEFAVE
Subj: POLYFORTH OBJECTS

I've uploaded the file POLYOBJ.ZIP. It contains the basic wordset from Dick Pountain's book *Object-Oriented Forth*. It will run only under polyFORTH/386. However, I believe it will also work under the 8086 version. I used `CELLS` and `+CELLS` whenever it was appropriate.

It's not particularly efficient, but it does work. I'm rewriting my database words to take advantage of objects, then I plan to work on execution speed.

—Gene

D.RUFFER [Dennis]

Gene, I've been doing a little bit of playing with your POLYOBJ file. It has a couple of problems with working on 8086 polyFORTH. I've tried to make the changes, but now the system isn't working anymore. It might be easier if you make a few changes and re-post it, since you know what makes it tick and how to check it. I'm just getting too lost in it to trust my changes anymore.

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

First, it is not recommended to change the function of existing words. Your change to the nucleus is okay, but I would call it another name instead of using an existing name. I changed your definition of -FIND to -FINDS and left the original -FIND alone.

Next, in quite a few places, you used 4 or 4+ where you should have used 1 CELL and CELL+. I made the changes, but I'm not sure I hit them all. Most of your stuff was okay, but blocks one and two were the worst cases. It is particularly difficult to know what to do with user variables, but the best bet is to use 1 CELLS instead of a literal number.

On the user variables, a relatively standard way to start them off is with the following statement:

```
STATE STATUS -  CELL+
\ last user offset
\ from block 198
```

Then, at the end of the block, you can use the following to let you know how many have been used:

```
CR .( Last USER =) .
```

You appear to have added the definition CELLS+ that is not standard. I've just returned that to CELLS + as it should have been.

The definition L@ uses a thing called -LINKS to remove some of the bits of the link field. That is only needed on the 386 version. I believe that L@ can be defined as AKA @ L@ for the 8086.

The definition SEAL is just too large and I get an "out of range" error on the /LOOP. I tried to refactor it, but I think that is where I messed up. Try your hand at seeing if you can keep the DO and the /LOOP a little closer together. I think I can see what you are doing there, but you can do a better job factoring it than I can, and you know how to test it.

That's as far as I got, but now the dictionary links are messed up after I load block seven. I believe the END> is the one that is doing it, which uses my butchered SEAL, but finding the problem is more than I want to deal with right now.

Thanks. —DaR

From: GENE LEFAVE

Sorry Dennis, I didn't realize just how incompatible with 8086 pF it was. I just got back from vacation so give me a few days to look it over.

Just for starters, the 17 CELLS used throughout is wrong. This should be CURRENT CONTEXT - (assuming that context precedes the threads and current follows). The 17 is from 16 threads plus CONTEXT.

I think a better solution would be a whole new version of -FIND that takes an address of a single dictionary thread and searches, fig-FORTH style. This would save a lot of trouble.

The problem with SEAL is probably related to the 17 CELLS. I don't recall offhand what is after CURRENT, but I'm sure setting it to zero will cause a lot a problems. SEAL scans down all the threads and creates a little mini-dictionary for the object.

I'll try to post a new version shortly.

—Gene

*   *   *

Object-oriented Forth has also been the subject of some regular Thursday night (9:30 p.m. Eastern time) open FIGGY BAR conferences.

To suggest an interesting on-line guest, leave e-mail posted to GARY-S on GEnie (gars on Wetware and the Well), or mail me a note. I encourage anyone with a message to share to contact me via the above or through the offices of the Forth Interest Group.

REFERENCE SECTION

Forth Interest Group
The Forth Interest Group serves both expert and novice members with its network of chapters, *Forth Dimensions*, and conferences that regularly attract participants from around the world. For membership information, or to reserve advertising space, contact the administrative offices:
Forth Interest Group
P.O. Box 8231
San Jose, California 95155
408-277-0668

Board of Directors
Robert Reiling, President *(ret. director)*
Dennis Ruffer, Vice-President
John D. Hall, Treasurer
Terri Sutton, Secretary
Wil Baden
Jack Brown
Mike Elola
Robert L. Smith

Founding Directors
William Ragsdale
Kim Harris
Dave Boulton
Dave Kilbridge
John James

In Recognition
Recognition is offered annually to a person who has made an outstanding contribution in support of Forth and the Forth Interest Group. The individual is nominated and selected by previous recipients of the "FIGGY." Each receives an engraved award, and is named on a plaque in the administrative offices.

1979 William Ragsdale
1980 Kim Harris
1981 Dave Kilbridge
1982 Roy Martens
1983 John D. Hall
1984 Robert Reiling
1985 Thea Martin
1986 C.H. Ting
1987 Marlin Ouverson
1988 Dennis Ruffer
1989 Jan Shepherd

ANS Forth
The following members of the ANS X3J14 Forth Standard Committee are available to personally carry your proposals and concerns to the committee. Please feel free to call or write to them directly:

Gary Betts
Unisyn
301 Main, penthouse #2
Longmont, CO 80501
303-924-9193

Mike Nemeth
CSC
10025 Locust St.
Glenndale, MD 20769
301-286-8313

Andrew Kobziar
NCR Medical Systems Group
950 Danby Rd.
Ithaca, NY 14850
607-273-5310

Elizabeth D. Rather
FORTH, Inc.
111 N. Sepulveda Blvd., suite 300
Manhattan Beach, CA 90266
213-372-8493

Charles Keane
Performance Packages, Inc.
515 Fourth Avenue
Watervleit, NY 12189-3703
518-274-4774

George Shaw
Shaw Laboratories
P.O. Box 3471
Hayward, CA 94540-3471
415-276-5953

David C. Petty
Digitel
125 Cambridge Park Dr.
Cambridge, MA 02140-2311

Forth Instruction
Los Angeles—Introductory and intermediate three-day intensive courses in Forth programming are offered monthly by Laboratory Microsystems. These hands-on courses are designed for engineers and programmers who need to become proficient in Forth in the least amount of time. Telephone 213-306-7412.

On-Line Resources
To communicate with these systems, set your modem and communication software to 300/1200/2400 baud with eight bits, no parity, and one stop bit, unless noted otherwise. GEnie requires local echo.

GEnie
For information, call 800-638-9636
- Forth RoundTable
  (ForthNet link)*
  Call GEnie local node, then type M710 or FORTH
  SysOps: Dennis Ruffer (D.RUFFER), Scott Squires (S.W.SQUIRES), Leonard Morgenstern (NMORGENSTERN), Gary Smith (GARY-S)
- MACH2 RoundTable
  Type M450 or MACH2
  Palo Alto Shipping Company
  SysOp: Waymen Askey (D.MILEY)

(Continued on page 34.)

Forth Interest Group Chapters

TALKING IT UP IN THE OUTFIELD

JACK WOEHR - 'JAX' on GEnie

A couple of issues ago in this column, we mentioned that we would soon start telephoning the various FIG Chapters located around North America in an attempt to re-establish contact between them and the central organization. The telephoning is under way as of this writing. I'm not sure what the results of these conversations have to do with the original goal of the calls, but it is good material for a series of magazine articles, of which this is the first.

Kansas City FIG Chapter
Chapter Coordinator: Linus Orth

The Kansas City, Missouri chapter no longer has regular meetings. They do, however, still have their mailing list together. Their most recent meeting was in December, 1989 when six members out of a mailing list of 90 showed up. The December meeting was the Kansas City chapter's first meeting in six months.

Mr. Orth feels that the chapter has achieved its original goal of putting all the local Forth programmers in touch with one another; they now know who they are and where they can be found, and often meet outside of the context of the Forth Interest Group. There are between twenty and thirty of these Forth programmers in the Kansas City area.

Mr. Orth doesn't participate in Forth telecom, but intends to do so sometime in the future. He would like to turn over his Chapter Coordinator responsibilities, but no individual has stepped forward to assume the post.

Mr. Orth feels that a printed handout for beginners that included a summary of learning materials would be a helpful tool that FIG could provide to local chapters.

Phoenix FIG Chapter
Chapter Coordinator: Dennis L. Wilson

The Phoenix chapter meets regularly on the fourth Thursday of each month. Charles Moore has spoken there a couple of times. Most of the meetings are open forums, with members comparing notes on their programming activities.

The Phoenix, Arizona chapter meets at the University of Arizona and is trying to attract more university students. Forth is taught occasionally at the University in the context of Artificial Intelligence.

Mr. Wilson would be interested in any efforts FIG makes to reach out to secondary education.

Give them the same exciting, leading-edge opportunity...

Huntsville FIG Chapter
Chapter Coordinator: Tom Konantz

The Huntsville, Alabama chapter of the Forth Interest Group has been inactive for about three years. Not enough people at meetings was the main problem. Mr. Konantz finds himself mostly programming in Ada these days. Mr. Konantz does not mind being listed in *Forth Dimensions* as a FIG contact person, however.

Houston FIG Chapter
Chapter Coordinator: Russell Harris

The Houston, Texas FIG Chapter meets on the third Monday of each month at 7:30 p.m. at the Houston Area League of Personal Computer Users (HAL-PC), 1200 Post Oak, Houston. Typical attendance ranges from six to 15 members. There is no organized activity currently; Forth classes held in the past had "no great success" in attracting attendance. Mr. Harris feels that the synergy of association with HAL-PC has been helpful in keeping the chapter active.

In the past, the chapter conducted a group project in which they wire-wrapped single-board computers based on the 1802, and had at them with the fig-FORTH listings for that processor. Lately, there has been a move for the chapter to "write its own Forth" as an exercise, but they are having trouble maintaining the necessary level of interest and active participation.

Mr. Harris finds that new attendees at meetings tend to be assembly-language programmers looking for a way to handle new hardware.

Mr. Harris finds the Harris Semiconductor RTX-2000 family of chips very exciting, and feels they could be a marvelous tool for revitalizing FIG. He notes that the "Golden Age" of FIG came at a time when Forth was an important tool for conquering new hardware, and that the RTX-2000 offers the same sort of challenge for the more sophisticated modern computer user.

Mr. Harris wonders aloud, even to the extent of phoning Harris Semiconductor himself, why Harris Semiconductor doesn't market the RTX-2001A Evaluation Board that they are using for the Harris/ESP RTX Design Contest at a break-even price, just to "seed" the programming world with experienced and eager RTX-2000 programmers. He thinks the national organization could attempt to encourage Harris Semiconductor in this direction, and that such a project available via mail order from the Forth Interest Group could revitalize local chapters by giving them the same sort of exciting, leading-edge opportunity that characterized the Forth Interest Group in bygone days.

More next issue.

TENTH ANNUAL ROCHESTER FORTH CONFERENCE ON EMBEDDED SYSTEMS

June 12th – 16th, 1990
University of Rochester
Rochester, New York

Invited Speakers

Mr. Charles Moore *Computer Cowboys*
Woodside, CA
ShBoom on ShBoom: A Microcosm of Hardware and Software Tools

Mr. Darrel Johansen *Orion Instruments, Inc.*
Menlo Park, CA
Using Forth to Analyze and Debug Kernel-less Embedded Systems

Dr. Rod Crawford *Microprocessor Engineering Ltd.*
Southampton, England
Active Messages and Passive Objects: An Object Oriented View of Forth

Dr. Kent Brothers *Stephenson Software*
Vancouver, BC Canada
The Forth System Behind VP-Planner: Designing for Efficiency in the Face of Complexity

Dr. Arne Henden *Ohio State University*
Columbus, OH
The Future of Forth in Astronomy

Dr. Sergei Baranoff *Leningrad Institute for Informatika*
Leningrad, USSR
Glasnost, Perestroika and Forth

AND... over 60 presentations on the state of the art in Forth and threaded interpretive languages, including comparisons of C, ADA and Forth for embedded systems. A one day seminar by Harris on the RTX family will precede the Conference.

Other areas of interest:

- Object oriented programming
- ANS X3J14 Forth Standard
- Working Groups
- Tour of the Laboratory for Laser Energetics
  See Fusion and Forth

Travel

USAir is the official air carrier of the Conference. For Conference travel between June 9th and June 19th arriving and departing from Rochester you can get fares of 40% off full coach, or 5% off the lowest applicable fare. For reservations or information please call *collect* (716) 244-9300, and ask for Ms. Barbara Jorden between 9:00 AM and 5:00 PM, Eastern Time.

Registration and Housing

The registration fee and university services includes all sessions, meals, and the Conference papers. Lodging is available at local motels or in the UR dormitories. Registration will be from 4-11 PM on Tuesday, June 12th in the Wilson Commons, and from 8 AM Wednesday, June 13th in Hubble Auditorium, where sessions will be held.

Name ____________________

Address ____________________

____

____

Telephone: Wk: (   ) ____________

Hm: (   ) ____________

Registration:

☐ $225.00

☐ $50.00 (full-time Students)

Total $ ________

University Services **$ 225.00**

Dormitory Housing, 5 nights

☐ $175.00 single

☐ $125.00 double

☐ Vegetarian Meal Option

☐ Non-smoking Roommate

☐ 5K Fun Run

☐ Harris RTX Seminar (June 12) **Call**

Total $ ________

Amount Enclosed $ ________

MC/Visa # ____________________ exp. ______

Please make checks payable to the **IAFR.** *Mail your registration to the Institute for Applied Forth Research Inc., Box A, 70 Elmwood Avenue, Rochester, NY 14611, USA.*

Phone orders: (716) 235-0168 / Fax: (716) 328-6426

(Continued from page 31.)

BIX (ByteNet)
For information, call 800-227-2983
- Forth Conference
  Access BIX via TymeNet, then type j forth
  Type FORTH at the : prompt
  SysOp: Phil Wasson (PWASSON)
- LMI Conference
  Type LMI at the : prompt
  Laboratory MicroSystems products
  Host: Ray Duncan (RDUNCAN)

CompuServe
For information, call 800-848-8990
- Creative Solutions Conference
  Type !Go FORTH
  SysOps: Don Colburn, Zach Zachariah, Ward McFarland, Jon Bryan, Greg Guerin, John Baxter, John Jeppson
- Computer Language Magazine Conference
  Type !Go CLM
  SysOps: Jim Kyle, Jeff Brenton, Chip Rabinowitz, Regina Starr Ridley

Unix BBS's with forth.conf (ForthNet links and reachable via StarLink node 9533 on TymNet and PC-Pursuit node casfa on TeleNet.)*
- WELL Forth conference
  Access WELL via CompuserveNet or 415-332-6106
  Fairwitness: Jack Woehr (jax)
- Wetware Forth conference
  415-753-5265
  Fairwitness: Gary Smith (gars)

PC Board BBS's devoted to Forth (ForthNet links)*
- East Coast Forth Board
  703-442-8695
  StarLink node 2262 on TymNet
  PC-Pursuit node dcwas on TeleNet
  SysOp: Jerry Schifrin
- British Columbia Forth Board
  604-434-5886
  SysOp: Jack Brown
- Real-Time Control Forth Board
  303-278-0364
  StarLink node 2584 on TymNet
  PC-Pursuit node coden on TeleNet
  SysOp: Jack Woehr

Other Forth-specific BBS's
- Laboratory Microsystems, Inc.
  213-306-3530
  StarLink node 9184 on TymNet
  PC-Pursuit node calan on TeleNet
  SysOp: Ray Duncan
- Knowledge-Based Systems
  Supports Fifth
  409-696-7055
- Druma Forth Board
  512-323-2402
  StarLink node 1306 on TymNet
  SysOps: S. Suresh, James Martin, Anne Moore
- Harris Semiconductor Board
  407-729-4949
  StarLink node 9902 on TymNet (toll from Post. St. Lucie)

Non-Forth-specific BBS's with extensive Forth Libraries
- Twit's End (PC Board)
  501-771-0114
  1200-9600 baud
  StarLink node 9858 on TymNet
  SysOp: Tommy Apple
- College Corner (PC Board)
  206-643-0804
  300-2400 baud
  SysOp: Jerry Houston

**ForthNet is a virtual Forth network that links designated message bases in an attempt to provide greater information distribution to the Forth users served. It is provided courtesy of the SysOps of its various links.*

(Continued from page 17.)

tient plus the remainder equals the dividend.

```
: T/   ( n1 -- index weight )
  #-199 MAX   #100 MIN
  #100 ONE-BASED/MOD
  NEGATE ;
```

Remarkably, there is an elegant division algorithm whose behavior with a negative divisor fits even better for this example.

Partition division is what calculates the number of trips a truck must take when the truck can hold, say, three boxes. Given 14 boxes, five trips are needed to carry the boxes; i.e., 14 partitioned by threes is five partitions. The remainder from a partition is inherently zero or negative, so that division by three gives remainders in the set {-2..0}. This represents the unused capacity of the truck—in the case of 14 boxes, the value is -1. Net result: 14 divided by three is five with the remainder -1.

Relating to the example, what matters is how this division is to behave with a negative divisor. The partition division with a positive divisor of three yields remainders in the set {-2..0}. With a negative divisor of -3 yielding remainders in the set {1..3}, we have an ideal division algorithm for this particular example. There is a concept here, concerning the relationship of these two sets of remainders, which for lack of a better term might be called "remainder continuity." Again note that, as usual, the results will conform to the division transformation.

The calculation for the example becomes:

```
: T/  ( n1 -- index weight )
  #-199 MAX   #100 MIN
  #-100 PARTITION/MOD ;
```

Is this optimal result merely some form of artifact, or is an underlying reality expressing itself? I don't know—although if I had to guess right now, I'd say artifact. This particular calculation is involved in setting gain bits in an amplifier board built with primitive TTL circuitry.

PS: Benefits of Symmetrical Division

During this review, I've encountered about 25 division cases in on-line code which would benefit by using or having floored division. I've found zero cases in which the symmetrical division algorithm had some benefit.

Robert Berkey has been a member of the Forth Interest Group since 1978 and can be reached on GEnie at the e-mail address R.BERKEY. He was a contributor to the Forth-83 Standard and is currently involved in the ANSI Forth standard's process. He is interested in examples of usefulness of rounded-to-zero integer division and integer division using negative divisors.

EMBEDDED CONTROL: PATH TO FORTH ACCEPTANCE

PHILIP KOOPMAN, JR. - WEXFORD, PENNSYLVANIA

Discussions on the Usenet Forth bulletin board and other arenas often concentrate on the theme of improving the acceptance of Forth among those not currently using it. I propose a strategy to accomplish this, which involves striving to make Forth the *language of choice* for embedded real-time control. This essay is meant to stir up discussion in the community about where we are really going, and to focus attention on why Forth should become more popular and how this may be done. It is intended as a starting point in a continuing discussion, not as a final word on the subject.

Why Worry About Forth Popularity?

Why are any of us worried about the popularity (or unpopularity) of programming in Forth? I would categorize the reasons for desiring that Forth become popular as follows:

• *The belief that Forth is a fundamentally better tool to solve problems.* Many of us believe that Forth is a fundamentally better way to solve problems. There are advantages to the interactive, incremental compilation environment used by most Forths. Many of us would like to see Forth in more widespread usage simply because we believe it is better.

• *Using Forth at the workplace.* Many of us write programs in Forth as a hobby or for minor projects, yet are unable to use Forth in our regular jobs. Others can use Forth for production programming, but only under unusual circumstances, usually involving tight deadlines or naive supervisors. One serious obstacle to using Forth is the lack of trained Forth programmers to maintain code. If Forth were a more socially acceptable language, such obstacles would be reduced and we could all use our favorite programming language for work as well as play.

• *Language support and development.* If Forth were to come into wider use, greater support would be available for it. The Forth market is small, so commercial Forth vendors can only afford to put relatively little effort into development and maintenance (compared to the resources used to support something like an Ada compiler). Even though Forth environments traditionally need much less support and maintenance than environments for other languages, the involvement of an order of magnitude more active Forth programmers would surely make available better tools, utilities, and environments.

This would create an installed base of applications and programmers.

• *Ego.* Many ardent Forth advocates want to receive praise eventually for their wisdom and foresight in choosing Forth as their favorite programming language. Forth programmers tend to be the best programmers (just ask them!).

But how compelling are these reasons? Are they sufficient to justify spending time and effort winning wider acceptance for the languages? Before we begin the attack, we should satisfy ourselves that Forth is worth fighting for. Gut feel and emotion are not enough for this; we must understand why we want Forth to succeed before we commit ourselves to the goal of having it succeed.

The Important of Being a Language of Choice

Forth is not a major force in the programming language scene. It is not taught in most schools. It is poorly represented in, or absent from, the computer section of bookstores. It is seldom presented at computer science and computer engineering conferences. It is not openly used in most companies. It is not used by a significant number of programmers in most disciplines. Yet, this lack of use is certainly not because of a lack of inexpensive software, nor for any other simple reason.

There is a long litany of reasons why Forth isn't used by more people. I won't indulge in an enumeration of why Forth is not popular now, since the list is well known. Most of the problems boil down to a lack of professional-quality development tools, lack of appreciation for the strengths and weaknesses of the language for different application areas, and the fact that it isn't already popular. (Most managers have a well-founded fear of using novel technology to solve problems that can be solved with familiar tools.)

One of the strengths of Forth is that it can be used in many applications. Since it is flexible and extensible, the language itself can be modified to incorporate many requirements. One of the schools of thought to gain greater acceptance for Forth is to extend the language so that it comes ready-made to solve any problem. Then, potential users will see the power of the language revealed, and will use it for all their programming needs.

This approach of developing "fat" Forth systems will not gain acceptance for Forth. Don't get me wrong—this is not to say that such Forth systems are bad or useless, rather to say that they are not the means to the end of Forth acceptance. The primary problem is not whether particular Forth standards or systems are "fat" or "thin." It is not what standard a Forth conforms to. It is not any technical reason at all.

The problem is one of marketing.

Ask almost any programmer what language is the best to use for numerical applications. The answer is Fortran. The best language for Unix-based applications is C. The best programming language for exploring language-implementation techniques and artificial intelligence is LISP. The best language for business programs on IBM mainframes is COBOL. The best language for a neophyte with a personal computer is BASIC (or, perhaps, LOGO for youngsters). Outside the Forth community, these statements will generally receive little argument.

Why can we say something like, "Fortran is the 'best' language for numerical applications" and find almost universal agreement? Because Fortran is the *language of choice* for those numerical applications. It is not necessarily the provably best language. It is the one that most people use. This means that there are libraries, development environments, programmers, and massive amounts of installed code which all presuppose the use of Fortran for certain classes of scientific and engineering applications.

Why not make Forth the *language of choice* for embedded real-time control applications? This would create an installed base of applications and programmers. It would also give an air of acceptability to Forth not only for embedded systems, but for other application areas as well.

The key is to market Forth as the best solution available to a restricted set of applications, not just as a good solution for everyone's problems. No one will believe that one language can do it all, but the idea of a flexible language which is well adapted to a single class of applications has considerable appeal. This is not to say that Forth shouldn't continue to evolve to support non-embedded applications, rather that such evolution should not compromise Forth's abilities in its major area of strength.

Embedded Control as a Target Application Area

There is a vacuum in the embedded-control marketplace. Until recently, almost all programming of small embedded-control systems (*e.g.*, 8051 and other eight-bit microcontroller systems) was done in assembly language. Now, with the advent of more powerful 16- and 32-bit controller chips, that is changing. More and more programmers are using or evaluating high-level languages to ease the burden of software development.

Embedded-control systems are the ideal application area for which to make Forth the language of choice. Their tightly constrained environments with demanding response requirements are ideally suited to the capabilities of classical Forth systems.

Forth has always had a strong foothold in embedded systems. But it has never been used by a majority. Because there is a vacuum and no concensus, a wide variety of programming languages are being pressed into service. The dominant language is C. This is not because C is inherently better for this application (or even, for that matter, very good at all). It is because that is the language most programmers are familiar with, having been trained in its use by universities and workstation-based programming projects.

If events are left to progress by themselves, C will become the language of choice for embedded control. Most of the user community will be happy with this, since C will be usable for many applications. Managers will be happy, since C has a lower risk and lower perceived cost than Forth. It always seems easier to pay the deferred, intangible costs for suboptimal programming environments than the immediate and concrete costs of programmer training to switch to a new language.

Large vendors of embedded control products and systems are not likely to change all this. They see the trend to using C, and will fall in line with the results of the market surveys and polls. Even makers of so-called "Forth chips" will probably support Forth just as an alternative language. Successful companies will have to spend most of their time and money supporting C to please their large number of C-based customers. C may well become the most-used language by popular demand, even if it makes suboptimal use of the hardware.

A Challenge

The reality of the situation is that the spark for making Forth the *de facto* standard language for embedded control will have to come from within the Forth community itself. Vendors and customers, if convinced of a trend towards Forth, will probably jump onto the bandwagon. But someone has to build the bandwagon, fuel it, and pilot it. The members of the Forth Interest Group are probably the only ones who can do this.

What is needed is a concerted marketing effort to promote Forth as the language of choice for embedded control, and in particular real-time control. This effort must incorporate articles in major periodicals (especially application-based articles), educational campaigns, and participation in mainstream events. For the purposes of Forth acceptance, one paper in a general computer application conference is worth two dozen at SIGForth, Rochester, or Asilomar. One article in *BYTE, IEEE Computer*, or *EDN* is worth an issue or two of *Forth Dimensions*. One application written in Forth in a company previously using C is worth a staff-full of FIG members at a Silicon Valley company (or a staff of SIGForth members at a Texas company). The point is to increase visibility, and make Forth look respectable to the outside world.

If we spend time and resources trying to convince programmers in the Unix world that they should be using Forth to write their window programs, or programmers on supercomputers that they should adapt LINPACK to run in a Forth environment, or if we are developing do-it-all Forth systems in hopes that outsiders will be impressed based on technical merit alone, we are wasting our time. And we have little time to waste. The window of opportunity is closing fast, driven by the optimizing C compilers and fancy development environments of the 32-bit RISC processors. As this window closes, not only will we lose our best chance to make Forth the language of choice, we may well lose the ability to sell Forth for these tasks at all, to most customers. So we need to act now, or resign ourselves to using a language that will forever be associated with APL and SNOBOL: neat ideas that never really caught on, but seem to never quite die, either.

None of this is meant to say that steps in this direction have not been taken in the past, nor to say that many facets of a successful Forth promotion are not already in place. However, the Forth community is internally fractured (or perhaps it was never unified). What is needed is a unity of purpose: a common vision. What we need is a champion to make it happen. We have not one but two special interest groups now: FIG and SIGForth. We also have a more academically oriented organization in the Institute for Forth Application and Research. Among these three organizations, we should be able to come up with a structure and method to seriously market Forth for the next year or two. If we don't, we had better brush up on our C and Fortran to feed our families in the coming years.

Philip Koopman Jr. is a senior scientist at Harris Semiconductor and an adjunct professor at Carnegie Mellon University. The opinions in this article are his, and do not necessarily reflect the views of Harris Semiconductor. Mr. Koopman presented this paper at the recent SIGForth meeting, and we publish it here in cooperation with ACM.

(Continued from page 10.)

amples. Screen Four illustrates the basic method for creating N2CONSTANT, the anonymous counterpart of the standard 2CONSTANT. Screen Five illustrates the basic method for creating the new classes 3CONSTANT and N3CONSTANT. In Screen Six, the alternative method is used to create 4CONSTANT and N4CONSTANT.

To maintain compatibility with the previously published version,[1] the words MAKEANON, ANON, ANON+, and STORESTACK are defined on Screen Seven.

Leonard Morgenstern is a retired pathologist and computer hobbyist. His interest in Forth goes back over ten years. Currently, he is a sysop of the Forth RoundTable on the GEnie network. His son, David Morgenstern, is also an author on computer-related subjects.

Screen 7 (Compatibility with previous version)

```
IMPERSONATOR ANON
: MAKEANON NVARIABLE IS [COMPILE] ANON ;
: ANON+ [COMPILE]
  [  [COMPILE] ANON  +  [COMPILE] LITERAL ] ;  IMMEDIATE

: STORESTACK ( n - ) 2* BOUNDS ?DO I ! 2 +LOOP ;
```

(Continued from page 13.)

-1 LOAD.

Note that (1) and (2) contain pieces of conditionals, and that the definition makers add IMMEDIATE after the definition to make ADD$ and DOS immediate words.

The cumbersome definition-makers of these examples could be made more useful by substituting a single character for the word ADD.BUF, for example:

```
: PREPARE
  0 BUF.POS !
  -1 BUF.ID !  ;

: MAKE.ADD$
  PREPARE
  % : ADD$ " (1)
  % (ADD$) " (2)
  % CONCAT THEN ; "
  % IMMEDIATE"  ;

: MAKE.DOS
  PREPARE
  % : DOS " (1)
  % (DOS) " (2)
  % >KEYBUF
    DOS.CALL THEN ;
  % IMMEDIATE "  ;
```

Chester H. Page earned his doctorate in mathematical physics at Yale and spent some 36 years at the National Bureau of Standards. His first Forth was Washington Apple Pi's fig-FORTH, which he modified to use Apple DOS, then ProDOS, and later to meet the Forth-79 and Forth-83 Standards. Recently, he added many features of F83, including a four-thread dictionary (but no shadow screens) and a vocabulary name format that provides for a search-order routine.

ADVERTISERS INDEX

Leo Brodie	12
CIBER	17
Dash, Find Associates	27
Embedded Systems Programming	40
Harvard Softworks	26
Institute for Applied Forth Research	33
Laboratory Microsystems	17
Miller Microcomputer Services	25
Next Generation Systems	30
Silicon Composers	2
Upper Deck Systems	27

FIG CHAPTERS

The FIG Chapters listed below are currently registered as active with regular meetings. If your chapter listing is missing or incorrect, please contact Kent Safford at the FIG office's Chapter Desk. This listing will be updated in each issue of *Forth Dimensions*. If you would like to begin a FIG Chapter in your area, write for a "Chapter Kit and Application." Forth Interest Group, **P.O. Box 8231, San Jose, California 95155**

U.S.A.

- **ALABAMA**
  Huntsville Chapter
  Tom Konantz
  (205) 881-6483

- **ALASKA**
  Kodiak Area Chapter
  Ric Shepard
  Box 1344
  Kodiak, Alaska 99615

- **ARIZONA**
  Phoenix Chapter
  4th Thurs., 7:30 p.m.
  Arizona State Univ.
  Memorial Union, 2nd floor
  Dennis L. Wilson
  (602) 381-1146

- **ARKANSAS**
  Central Arkansas Chapter
  Little Rock
  2nd Sat., 2 p.m. &
  4th Wed., 7 p.m.
  Jungkind Photo, 12th & Main
  Gary Smith (501) 227-7817

- **CALIFORNIA**
  Los Angeles Chapter
  4th Sat., 10 a.m.
  Hawthorne Public Library
  12700 S. Grevillea Ave.
  Phillip Wasson
  (213) 649-1428

  North Bay Chapter
  2nd Sat., 10 a.m. Forth, AI
  12 Noon Tutorial, 1 p.m. Forth
  South Berkeley Public Library
  George Shaw (415) 276-5953

  Orange County Chapter
  4th Wed., 7 p.m.
  Fullerton Savings
  Huntington Beach
  Noshir Jesung (714) 842-3032

  Sacramento Chapter
  4th Wed., 7 p.m.
  1708-59th St., Room A
  Bob Nash
  (916) 487-2044

  San Diego Chapter
  Thursdays, 12 Noon
  Guy Kelly (619) 454-1307

  Silicon Valley Chapter
  4th Sat., 10 a.m.
  H-P Cupertino
  Bob Barr (408) 435-1616

  Stockton Chapter
  Doug Dillon (209) 931-2448

- **COLORADO**
  Denver Chapter
  1st Mon., 7 p.m.
  Clifford King (303) 693-3413

- **CONNECTICUT**
  Central Connecticut Chapter
  Charles Krajewski
  (203) 344-9996

- **FLORIDA**
  Orlando Chapter
  Every other Wed., 8 p.m.
  Herman B. Gibson
  (305) 855-4790

  Southeast Florida Chapter
  Coconut Grove Area
  John Forsberg (305) 252-0108

  Tampa Bay Chapter
  1st Wed., 7:30 p.m.
  Terry McNay (813) 725-1245

- **GEORGIA**
  Atlanta Chapter
  3rd Tues., 7 p.m.
  Emprise Corp., Marietta
  Don Schrader (404) 428-0811

- **ILLINOIS**
  Cache Forth Chapter
  Oak Park
  Clyde W. Phillips, Jr.
  (312) 386-3147

  Central Illinois Chapter
  Champaign
  Robert Illyes (217) 359-6039

- **INDIANA**
  Fort Wayne Chapter
  2nd Tues., 7 p.m.
  I/P Univ. Campus
  B71 Neff Hall
  Blair MacDermid
  (219) 749-2042

- **IOWA**
  Central Iowa FIG Chapter
  1st Tues., 7:30 p.m.
  Iowa State Univ.
  214 Comp. Sci.
  Rodrick Eldridge
  (515) 294-5659

  Fairfield FIG Chapter
  4th Day, 8:15 p.m.
  Gurdy Leete (515) 472-7077

- **MARYLAND**
  MDFIG
  Michael Nemeth
  (301) 262-8140

- **MASSACHUSETTS**
  Boston Chapter
  3rd Wed., 7 p.m.
  Honeywell
  300 Concord, Billerica
  Gary Chanson (617) 527-7206

- **MICHIGAN**
  Detroit/Ann Arbor Area
  Bill Walters
  (313) 731-9660
  (313) 861-6465 (eves.)

- **MINNESOTA**
  MNFIG Chapter
  Minneapolis
  Fred Olson
  (612) 588-9532

- **MISSOURI**
  Kansas City Chapter
  4th Tues., 7 p.m.
  Midwest Research Institute
  MAG Conference Center
  Linus Orth (913) 236-9189

  St. Louis Chapter
  1st Tues., 7 p.m.
  Thornhill Branch Library
  Robert Washam
  91 Weis Drive
  Ellisville, MO 63011

- **NEW JERSEY**
  New Jersey Chapter
  Rutgers Univ., Piscataway
  Nicholas Lordi
  (201) 338-9363

- **NEW MEXICO**
  Albuquerque Chapter
  1st Thurs., 7:30 p.m.
  Physics & Astronomy Bldg.
  Univ. of New Mexico
  Jon Bryan (505) 298-3292

- **NEW YORK**
  Rochester Chapter
  Odd month, 4th Sat., 1 p.m.
  Monroe Comm. College
  Bldg. 7, Rm. 102
  Frank Lanzafame
  (716) 482-3398

- **OHIO**
  Cleveland Chapter
  4th Tues., 7 p.m.
  Chagrin Falls Library
  Gary Bergstrom
  (216) 247-2492

- **Columbus FIG Chapter**
  4th Tues.
  Kal-Kan Foods, Inc.
  5115 Fisher Road
  Terry Webb
  (614) 878-7241

  Dayton Chapter
  2nd Tues. & 4th Wed., 6:30 p.m.
  CFC. 11 W. Monument Ave. #612
  Gary Ganger (513) 849-1483

- **OREGON**
  Willamette Valley Chapter
  4th Tues., 7 p.m.
  Linn-Benton Comm. College
  Pann McCuaig (503) 752-5113

- **PENNSYLVANIA**
  Villanova Univ. Chapter
  1st Mon., 7:30 p.m.
  Villanova University
  Dennis Clark
  (215) 860-0700

- **TENNESSEE**
  East Tennessee Chapter
  Oak Ridge
  3rd Wed., 7 p.m.
  Sci. Appl. Int'l. Corp., 8th Fl.
  800 Oak Ridge Turnpike
  Richard Secrist
  (615) 483-7242

- **TEXAS**
  Austin Chapter
  Matt Lawrence
  PO Box 180409
  Austin, TX 78718

  Dallas Chapter
  4th Thurs., 7:30 p.m.
  Texas Instruments
  13500 N. Central Expwy.
  Semiconductor Cafeteria
  Conference Room A
  Clif Penn (214) 995-2361

  Houston Chapter
  3rd Mon., 7:30 p.m.
  Houston Area League of PC Users
  1200 Post Oak Rd.
  (Galleria area)
  Russell Harris
  (713) 461-1618

- **VERMONT**
  Vermont Chapter
  Vergennes
  3rd Mon., 7:30 p.m.
  Vergennes Union High School
  RM 210, Monkton Rd.
  Hal Clark (802) 453-4442

- **VIRGINIA**
  First Forth of Hampton Roads
  William Edmonds
  (804) 898-4099

  Potomac FIG
  D.C. & Northern Virginia
  1st Tues.
  Lee Recreation Center
  5722 Lee Hwy., Arlington
  Joseph Brown
  (703) 471-4409
  E. Coast Forth Board
  (703) 442-8695

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

- **WISCONSIN**
  Lake Superior Chapter
  2nd Fri., 7:30 p.m.
  1219 N. 21st St., Superior
  Allen Anway (715) 394-4061

INTERNATIONAL

- **AUSTRALIA**
  Melbourne Chapter
  1st Fri., 8 p.m.
  Lance Collins
  65 Martin Road
  Glen Iris, Victoria 3146
  03/29-2600
  BBS: 61 3 299 1787

  Sydney Chapter
  2nd Fri., 7 p.m.
  John Goodsell Bldg., RM LG19
  Univ. of New South Wales
  Peter Tregeagle
  10 Binda Rd.
  Yowie Bay 2228
  02/524-7490
  Usenet
  tedr@usage.csd.unsw.oz

- **BELGIUM**
  Belgium Chapter
  4th Wed., 8 p.m.
  Luk Van Loock
  Lariksdreff 20
  2120 Schoten
  03/658-6343

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

- **CANADA**
  BC FIG
  1st Thurs., 7:30 p.m.
  BCIT, 3700 Willingdon Ave.
  BBY, Rm. 1A-324
  Jack W. Brown
  (604) 596-9764
  BBS (604) 434-5886

  Northern Alberta Chapter
  4th Sat., 10a.m.-noon
  N. Alta. Inst. of Tech.
  Tony Van Muyden
  (403) 486-6666 (days)
  (403) 962-2203 (eves.)

  Southern Ontario Chapter
  Quarterly, 1st Sat., Mar., Jun., Sep., Dec., 2 p.m.
  Genl. Sci. Bldg., RM 212
  McMaster University
  Dr. N. Solntseff
  (416) 525-9140 x3443

- **ENGLAND**
  Forth Interest Group-UK
  London
  1st Thurs., 7 p.m.
  Polytechnic of South Bank
  RM 408
  Borough Rd.
  D.J. Neale
  58 Woodland Way
  Morden, Surry SM4 4DS

- **FINLAND**
  FinFIG
  Janne Kotiranta
  Arkkitehdinkatu 38 c 39
  33720 Tampere
  +358-31-184246

- **HOLLAND**
  Holland Chapter
  Vic Van de Zande
  Finmark 7
  3831 JE Leusden

- **ITALY**
  FIG Italia
  Marco Tausel
  Via Gerolamo Forni 48
  20161 Milano
  02/435249

- **JAPAN**
  Japan Chapter
  Toshi Inoue
  Dept. of Mineral Dev. Eng.
  University of Tokyo
  7-3-1 Hongo, Bunkyo 113
  812-2111 x7073

- **NORWAY**
  Bergen Chapter
  Kjell Birger Faeraas,
  47-518-7784

- **REPUBLIC OF CHINA**
  R.O.C. Chapter
  Chin-Fu Liu
  5F, #10, Alley 5, Lane 107
  Fu-Hsin S. Rd. Sec. 1
  TaiPei, Taiwan 10639

- **SWEDEN**
  SweFIG
  Per Alm
  46/8-929631

- **SWITZERLAND**
  Swiss Chapter
  Max Hugelshofer
  Industrieberatung
  Ziberstrasse 6
  8152 Opfikon
  01 810 9289

- **WEST GERMANY**
  German FIG Chapter
  Heinz Schnitter
  Forth-Gesellschaft C.V.
  Postfach 1110
  D-8044 Unterschleissheim
  (49) (89) 317 3784
  Munich Forth Box:
  (49) (89) 725 9625 (telcom)

SPECIAL GROUPS

- **NC4000 Users Group**
  John Carpenter
  1698 Villa St.
  Mountain View, CA 94041
  (415) 960-1256 (eves.)

THREE DAYS OF PROFESSIONAL GROWTH . . .

ATTEND THE SECOND ANNUAL EMBEDDED SYSTEMS CONFERENCE

THE EMBEDDED SYSTEMS CONFERENCE IS your best source of objective, up-to-date information on embedded systems programming. The conference is growing in 1990 — and you can grow with it.

IN STATE-OF-THE-ART LECTURES and in small intensive workshops, you'll learn real-world solutions that will immediately increase your productivity. You'll be learning from the best available source — the most widely respected practitioners of the art and science of microprocessor and microcontroller-based development.

> ***"I thought the conference was excellent! It was the best conference/seminar I have ever attended."***
> **RANDY KREMSKE, AMERSHAM CORP., ARLINGTON HEIGHTS, IL**

Facing expanding development pressures? Respond with finesse.

FOR YEARS YOU'VE BEEN SUBJECTED to so-called objective panel discussions that turned out to be long-winded oral press releases from vendors. The Embedded Systems Conference is different. In-depth practical workshops will be headed by well-known and respected practitioners, like P.J. Plauger, Ray Duncan, Larry Constantine, Michael Slater, and Robert Ward, who will give you knowledge that you can take back and apply in your job today!

> ***"This conference did the best job of any that I have attended of fulfilling my expectations and needs. The mix and breakdown of lectures and workshops worked well."***
> **TIMOTHY CROSS, CATERPILLAR TECH CENTER, PEORIA, IL**

YOU'LL BE ABLE TO CHOOSE FROM 70 lectures and workshops. . .hands-on tutorials that get down to the nittiest, grittiest levels of designing, coding, debugging, and building embedded systems.

> ***"Overall, I was very impressed. I would definitely recommend this conference to anyone involved in embedded systems programming."***
> **GREG TOTH, NORTHROP ELECTRONICS, HAWTHORNE, CA**

The Marketplace for Productivity Tools

IF YOU'VE KNOWN THE FRUSTRATION of shopping for productivity tools designed especially for the embedded developer — look no further. At the Embedded Systems Conference Product Exhibit you'll find the marketplace for emulators, single-board computers, debugging tools, languages, and real-time operating systems.

AS AN ATTENDEE TO THE EMBEDDED SYSTEMS CONFERENCE you'll learn how to increase your productivity, how to select the right tool for the job, and how to get more out of the tools you already use.

AND YOU'LL GET IT ALL IN THREE DAYS. It's your best investment in productivity.

EMBEDDED SYSTEMS CONFERENCE

Sept. 25-28, Hyatt Regency, San Francisco Airport

Sponsored by: Embedded Systems Programming and Software Development Seminars, a division of Miller Freeman Publications, 500 Howard St., San Francisco, CA 94105.

For more information call Lynne Mariani at (415)995-2471 fax: (415)995-2494

Forth Interest Group
P.O.Box 8231
San Jose, CA 95155

Second Class
Postage Paid at
San Jose, CA