VOLUME XI, NUMBER 5 JANUARY/FEBRUARY 1990

FORTH DIMENSIONS

CONTROLLING REGULAR EVENTS

DOUBLE-ENTRY BOOKKEEPING

DEVELOPING A STEP TRACE

BINARY TABLE SEARCH

SEEING FORTH

F O R T H

D I M E N S I O N S

EDITORIAL

The FIGGY award is presented by the Forth Interest Group to those whose efforts have contributed significantly to the Forth community. Jan Shepherd was honored in 1989, joining the ten previous recipients, whose names are engraved on a plaque in the administrative offices and are listed in the "Reference Section" in this magazine. Jan heads the management team that takes care of FIG's daily business, the Association Development Center of San Jose, California. Anyone who knows her can attest that Jan is at every late-night meeting, convention booth, and crisis intervention. She and her staff have always been willing to outdo themselves on behalf of FIG and *Forth Dimensions*.

While you are browsing the "Reference Section," you may note some changes. The list of on-line resources has been updated significantly, so be sure to revise your auto-dial instructions! And when you are on-line, be sure to leave a personal note to the SysOps. BBS's are very interactive places, and the people running them not only expect but need your input. Even if you aren't uploading lots of files or joining various debates, let the SysOps know you appreciate their efforts and tell them about the things you like or dislike.

The autumn of each year brings a tradition to the Forth community: the annual FORML conference. It is, perhaps, the most venerable Forth institution and the least well known; it may also be the most intimidating, especially when it comes to exposing your ideas to the intimate assemblage of master-level Forth programmers. The most recent FORML was a sold-out affair, and long-time participant Peter Midnight is preparing a report for us which will be appearing shortly. The published proceedings look heftier than last year's edition; when it is available, you will find it on the FIG mail order form in these pages.

While we were preparing this issue, word came that readers from around the world were preparing articles about Forth hardware. You will remember our call for articles on that subject earlier this year, in which we offered payment for the top three chosen by the referees. The promising pile of manuscripts on my desk has been growing, with more due by the encroaching deadline. Our editorial work is cut out for us, and you will be able to see the results in our next issue—see you then!

—Marlin Ouverson
Editor

Forth Dimensions
Published by the
Forth Interest Group
Volume XI, Number 5
January/February 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.

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

Bad Press and *Still* Unknown

Dear Marlin,

I finally decided to learn more about sorting and tackle Quicksort. Of course, I would use Forth to rapidly play around with the algorithm and write some neat displays. Hah! I perused past articles in *FD* to learn how to do it, and was stymied by Forth's biggest weakness: it seems to encourage unreadable coding.

For example, in *FD* V/5 page 29, I was nipped by the phrase:

```
SWAP ROT 2OVER 2OVER - ROT ROT
- < IF 2SWAP THEN
```

And in *FD* VI/5 page 29, I was duped by the phrase:

```
ROT DROP DUP 2 PICK 2 PICK 2
PICK = = AND
```

Is it any wonder why Forth is still relatively unknown and gets bad press? Sure, other languages have their own confusing aspects. For example, the phrase:

```
(*(void(*)())0 )();
```

means something in C. But such horrors usually will not find their way into a beginners' text.

I looked at Quicksort in popular languages. Then I experimented to see how easy it is to translate the algorithm into Forth. The enclosed listing is an almost word-for-word translation of Quicksort written in the C language. Even the control structures were translated. The big drawback here is the prefix `L`, but I submit that it is more readable and maintainable than what was found in Forth. To add a running dump, for example, just add the phrase:

```
L i L j L nr L pivot
WORD.TO.SHOW.STATE
```

In contrast, with Macho Stack Pumping it would require a rewrite to thrash the values into place.

Another cause of unreadable code is the use of screens to store source. It is not natural. People are raised to see an 8.5" x 11" paper as the natural size to hold words. This is started in school, and is maintained at work and even in personal correspondence. Further, modern word processors are evolving into WYSIWYG page designers. The fact that most systems cannot display a whole page is temporary. The fact that a language insists on dividing source into half-pages is medieval. This adds to the unreadableness of Forth, because too many times code and comments must be crammed to fit into half a page.

As F-PC and other Forths have shown, interactive loading of source is still possible with stream source. In F-PC, one can load a stream file starting at any line. This allows a fast edit/test cycle, as in block-oriented Forths.

[Earlier this year,] a magazine had an article written by the owner of a Forth language supplier. He wrote C code to create Intel Hex Format files. I wonder, if he had used Forth, would it have been transportable, readable, and simple? Would the magazine have even published it?

Sincerely,

Jose Betancourt
Sunnyside, New York

```
\   QuickSort in F-PC using Parameter Stack Frames     by Jose Betancourt

: QuickSort   (  adr.of.array, #elements )          \  recursive QuickSort.
   DUP 2 <     \ Is #elements less than two?
   IF      2DROP   \ exit sort, cannot be repartitioned.
   ELSE        ( sort this partition. )
     L(  adr ne \ i j temp nr pivot done )     \  create parameter frame.
     A[Ø] @ is temp     L adr  A[Ø] !          \    point to array start.
     L ne    2 /    A[@]   IS pivot            \     pick middle element.
     -1 IS i  FALSE IS done   L ne IS j  \ initialize pointers and flag.
     BEGIN      \ partition into two parts.
        BEGIN                            \  find first element to move right.
               ++ i   L i A[@]   L pivot   <  WHILE
        REPEAT
        BEGIN                             \  find first element to move left.
               -- j   L j A[@]   L pivot   >  WHILE
        REPEAT
        L i  L j  >=     \  have the boundaries met ?
        IF      TRUE IS done
        ELSE      L i  L j     EXCHANGE[]     \ i and j elements.
        THEN        L done    \ partitions made?
     UNTIL     L ne  L i  -   IS nr   \ number of elements in right side.
     L i  L ne   2 /  <     \ sort smallest partition first.
     IF    ( first left side )        L adr   L i   RECURSE
           ( then right )          L i  A[&]   L nr  RECURSE
     ELSE    ( first right side )     L i  A[&]   L nr  RECURSE
             ( then left )                L adr   L i   RECURSE
     THEN
     L temp   A[Ø] !   S()      \ reset array pointer and kill frame.
   THEN     ;   \ End QuickSort

/*
     Pre and post incrementing and decrementing local prefixes would
be more compact and parallel the original language code.  For example,
L++ can fetch a local variable then increment the value stored there,
whereas, ++L, can increment then fetch the value.  Thus, the phrase
"++ i L i A[@]"   can be written as "++L i  A[@]".
*/
\S
```

(Screens continued on next page.)

Screens Foreshadowed (but not shadow screens)

Dear Marlin,

After seeing your response to Robert Hoffpauer and me on the source of "The rest is silence," [Letters, *FD* X/6), you *deserve* to get this...

Shakespeare made a broad mark on the development of the English language. It's not widely known just how far ahead of his time he really was. I found he had penned this little far-seeing dedication to a mysterious Mr. W.H. (who has never been unambiguously identified) into the first edition of his sonnets in 1609:

> "To the onlie begetter of
> The insung sonnets
> Mr. W.H. all happinesse
> And that eternitie
> Promised
> by
> Our ever-living poet
> Wisheth
> The well-wishing
> Adventurer in
> Setting
> Forth"

What could he have been doing, writing things with a title line, skip a line, a fourteen-line structure. Did they have blocks back then?

Glenn Toennes
843 Maywood
Escondido, California 92027

Only writer's block.—Ed.

(Betancourt screens, continued.)

```
\ -------------------- E X A M P L E ------------------------------

\ one way of defining the array access words

CREATE  VECT@     ' NOOP ,   ' C@ ,   ' @ ,            \ array of @ vectors.
CREATE  VECT!     ' NOOP ,   ' C! ,   ' ! ,            \ array of ! vectors.

VARIABLE   TYPE[]                                      \ bytes per cell
: V@      TYPE[] @  2*  VECT@ +  @ EXECUTE ;           \ vectored fetch.
: V!      TYPE[] @  2*  VECT! +  @ EXECUTE ;           \ vectored store.

VARIABLE A[Ø]                                          \ pointer to Øth cell.
: A[&]     ( ndx -- adr)   TYPE[] @  *  A[Ø] @ + ;     \ array pointer.
: A[@]     ( ndx -- m)     A[&]  V@ ;                  \ array fetch.
: A[!]     ( n  ndx)       A[&]  V! ;                  \ array store.

: EXCHANGE[]         ( i j ---  )     \ exchange elements i and j of array.
    L( i j )   L i  A[@]    L j  A[@]   L i  A[!]   L j  A[!]   S()  ;

\   testing words.

CREATE storage  1 , 15 , 8 , 2 , 3 , 5 , 1Ø , 4 , 7 , 9 , 6 , ( 11 cells)

2 CONSTANT integer

: showarray      11 Ø  DO   I  A[@]   3 .R    LOOP ;

: TESTSORT
       storage  A[Ø] !   integer  TYPE[] !   CR   showarray
       Ø A[&]  11  QuickSort     CR    showarray   ;

\ End example.
```

Null Strings, Count Too!

Dear Sir,

I once encountered in print a rationalization of the null-delimited form of string. The author claimed the immense benefit of "...being able to operate on the string without having to know how long it is." This is claiming a virtue out of a feature you don't have anyway. Charles Moore did this when he disdained the use of floating-point arithmetic. There are cases where the null-delimited form is vital. Anybody who passes strings to MS-DOS, for example, must do so in ASCIIZ, which uses the null-delimited format. Most users seem to do their work in standard Forth format and define a word to perform the conversion as required.

A simple alternative is available, however. This is to specify Forth strings to have a leading count byte *and* a trailing null. [See Figure One.] Thus, no special words are needed to pass a string parameter to MS-DOS. The necessary changes to the kernel are quite small. The new string definition is almost upwardly compatible from the original.

A more radical change (not yet implemented) would use a 16-bit count field and explicitly limit the maximum length of strings.

Yours faithfully,
J.D. Huttley
19 Duncan Avenue
Te Atatu Sth.
Auckland 8
New Zealand

A Fast Thousand Primes

Dear Sirs,

I have enclosed a Forth program for possible inclusion into your magazine. The title is "Primes," and it will compute the first 1000 prime numbers. It takes a little over two minutes to do this.

It was written on an Amiga 500 using jForth. This is a good choice, as the single-length numbers are 32 bits. jForth also allows double-length (64 bits) numbers.

The mode of operation is to maintain two lists of numbers: `primes` keeps track of the prime numbers as they are

Primes Listing

Two sets of numbers are recorded. One is the primes, the other is the squares of the primes. The primes are used as the source of divisors. Note that the form:

```
: word ( 10 3 ... 10 6 )
       ( 10 12 ... 10 6 )
  ...  ;
```

is used to show various types of stack data and the results of each type of stack input.

```
1100  array primes   \ create 1100 slots
1100  array primes^2 \

variable pointer*

8100  constant max-integer

variable integer

: cleararray 1100 0 do 0 i primes ! 0 i primes^2 ! loop ;

: flag-loop  0 ;
: flag-repeat  0 ;
: flag-done  1 ;
: flag-true  -1 ;
: flag-false  0 ;
```

(Continued on next page.)

discovered; `primes^2` keeps track of their squares. To check any number, a fraction is made by putting that number over each of a series of prime numbers. The primes checked begin with two and may continue to the square root of the next prime over the number being tested. Rather than performing a square root operation, the table of squares of primes is used. If the process of looking for a divisor of a given number cycles through all the lesser primes and arrives at one whose square is larger than the number, the process is stopped and the number is deemed to be prime.

The process of division is replaced by a subtraction process. This (hopefully) is faster than division. It is done by doubling the denominator and checking to determine if the new denominator is larger than the numerator. If not, it is again doubled, repeatedly, until it is larger than the numerator. It is then divided in half to reduce it to less than the numerator, and this new denominator is now subtracted from the numerator. This process determines a new numerator.

The process is continued until either a zero is arrived at, showing that the number is not prime; or a proper fraction is arrived at, showing that the next prime must be picked from the list and tried. The lists `primes` and `primes^2` are double purpose, in that new numbers are added to the list and old numbers are chosen off the list and used in the search for new primes.

It would be interesting to find a fast way to square the prime numbers, as the other operations (doubling and subtracting) are well-suited to assembly language programming. Perhaps someone would be interested in speeding this up more by using assembly.

Yours truly,
Allan Rydberg
RFD #1, Box 46C
Sterling, Connecticut

Count	0 or more bytes	$00
N	N + 1	

Figure One. Pass N+1 to MS-DOS as the string's start address.

(Continued.)

```
  : copy2  over over ;
  : double  1 ashift ;
  : subtract  - ;
  : half -1 ashift ;
  : square  dup * ;

   : setup  1 pointer* ! 2 1 primes ! 4 1 primes^2 ! ;

   : double-denom
               ( 10 3 ... 10 6 0-flag )     \  10/3 = 0flag or
               ( 10 6 ... 10 12 1 flag )  \    10/6 = 1flag
     copy2 > if double  flag-loop
             else copy2 < if half flag-done
                              else copy2 = if flag-done
                                           else ." error in double-denom "
                                     then
                               then
             then ;
 \ double-denom copy top pair of stack
 \                if 10 > 3 then double +flag for loop
 \                else if 10< 3 then divide by 2 and flag done
 \                     else if 10=3 then flag-done
 \                             else print error message

  : max-denom  ( 10 3 ... 10 6 )            \ loop to find max denominator
     begin double-denom until ;
\  max-denom =  loop to double-denom and continue until flag-done is returned

  : test-done ( 1 3 ... 1 3 flag-true )
             ( 10 3 ... 10 3 flag-false )
             copy2  < if  flag-true  else flag-false
                    then ;
 \ test-done  copy2 and see if num < denom if true then  flag true

  : test-equal ( 10 3 ... 10 3 flag-false )  ( 3 3 ... 0 3 flag-true )
             copy2  = if copy2
                         subtract  rot drop swap flag-true
                      else flag-false
                      then ;
 \ test-equal if equal leave remainder,denomerator,flag

 : subtract-d-from-n     ( 10 3 10 6 ... 4 3 0 )
                 subtract over rot drop rot drop 0 ;
     \ subtract-d-from-n subtract 6 from 10 leave 4,3,0

 : cycle  ( 10 3 ... 4 3 )  ( 10 5 10 10 )
    test-done  if  flag-done
               else test-equal if flag-done
                               else copy2  max-denom
                                    subtract-d-from-n
                               then
               then ;

 : test-fraction  ( 10 3 ...   1 )  ( 10 2 ... 0 )
    begin cycle until drop ;
 \ test-fraction will leave the remainder of any division on the stack

  : increment   pointer* @ 1 + pointer* ! ;

  : new-prime
            integer @ dup  increment  pointer* @  rot over . .
            primes !
            integer @ square  pointer* @  primes^2 ! ;

 : test-answer 0 = ;

 : test-for-zero  ( i ... F )
              0  = if flag-true else flag-false then ;

 : create-denominators
             integer @ 1 do
                          integer @  i primes @ dup
                           test-for-zero
                            if drop drop  new-prime leave
                            else  integer @ i primes^2 @ <
                                 if drop drop new-prime leave
                                 else
                                 test-fraction test-answer if
                                 leave then
                                 then
                            then
                         i  integer @ 1 -  =  if new-prime then
                         loop ;

 : run*
          \ cycle from 3 thru max-integer and set each # equal to integer
          \ then jump to create-integer
            cleararray  setup
            max-integer  3 do
            i integer !  create-denominators
            loop ;
```

F83

DOUBLE-ENTRY BOOKKEEPING

J.J.MARTENS - KAUKAUNA, WISCONSIN

In its present form, DE-BOOKS is a capsulated version of my personal bookkeeping system. If the reader is totally unfamiliar with double-entry bookkeeping, I suggest some research in this area. It's a little tricky but, like Forth, can be very rewarding once you get the hang of it.

Although most people may not want to bother with a personal bookkeeping system, it's a rare individual who doesn't have occasional use for a financial statement. DE-BOOKS was conceived as the former, but in the development I discovered the latter emerges as a by-product and—for some folks—may well be the tail that wags the dog. Assuming that you have looked over the code and explanatory material in the shadow screens, let's touch on a few of the details.

Screen 15 is the only shadow necessary to the operation of the program. Ordinarily, this screen contains the user's personal account categories, but until you're familiar with the operation it may be wise to use the working accounts in the order provided. Important information regarding account names and numbers is in screen 17.

Screen 18 is my favorite. If the arrays are the body of the system, this must be the heart. It doesn't look like much, but it may be where I learned the meaning of iteration. Early versions used up to three screens.

The next four screens represent the goals we are trying to reach. If one can draw a line between bookkeeping and accounting, it may be here, between the trial balance (screen 9) and the beginning of the financial statement (screen 10). The program produces one as easily as the other. I like to think of it as Cinderella the bookkeeper being transformed into Ms. Financial Statement the accountant, via the magical power of Forth.

The transitory (P&L) in screen 11 is unique in that we never add to it, subtract from it, or clear it. We just store (screen 11, line 12) and fetch (screen 12, line 11). Any profit or loss determined by the program is a reflection of the journal or ledger at the instant the financial statement is taken.

Trial balances and financial statements are taken as desired.

We could use the stack instead of the variable to accomplish this, if desired. In retrospect, that may be a better way of doing it. If we used the stack, the balance sheet could precede profit-and-loss on the financial statement, and the actual profit or loss would be on the stack for RECAP. We live and learn. Better late than never. The old clichés can be comforting. On with the show!

In my personal version, the MS-DOS COMMAND.COM, F83.COM, and DE-BOOKS.BLK are permanently on the disk. NEWBOOKS is used to set up the original account balances, and TRANSFER puts them in the ledger. The F83 word SAVE-SYSTEM is used to save the opening balances as a command file. This setup is done once.

At the end of the month, the command file is run on DE-BOOKS.BLK, and the deposits and checks for the month are posted to the JOURNAL, making sure that the debits and credits balance. TRANSFER adds the current month's data to the ledger, and SAVE-SYSTEM creates a new command file. This routine is repeated monthly.

Trial balances and financial statements are taken as desired and the older command files are erased as the disk fills. Hard copy is a must but, of course, that's another story. The version I use includes printing utilities for an Epson LX-86 printer.

My references include the source code for F83; *Inside F83* by C.H. Ting, Ph.D.; *Starting Forth* and *Thinking Forth* by Leo Brodie, FORTH, Inc.; *Mastering Forth* by Anita Anderson and Martin Tracy, Micro-Motion; and *Forth Dimensions*.

J.J. Martens ran the family business for nearly three decades, then spent several self-employed years until his 'practical retirement.' His interest in Forth and subsequent purchase of a Jupiter Ace computer (and more equipment later), was aroused in 1983 by Popular Computing, *which he calls "...a good magazine, now extinct." To those who find double-entry bookkeeping more difficult than Forth, he offers Edmund Burke's advice, "Don't despair—but if you do, work on in despair!"*

```
     0
0 Double-entry bookkeeping. 15 shadows--0/15, etc.      10-20-88jm
1     DE-BOOKS is a bare-bones double-entry bookkeeping system
2 that uses the Laxen-Perry F83 implementation of the Forth-83
3 Standard.
4
5     It consists of a general journal, a general ledger, a
6 mechanism for posting original entries to either, and a word to
7 transfer journal data directly to the ledger.  A trial balance
8 and or a simple financial statement can be taken from either
9 journal or ledger at any time.
10
11    The working chart of accounts in screen 15 can accommodate
12 48 account categories and can easily be edited to suit the user.
13
14    The application was written with a Radio Shack Tandy 1000
15 computer over an MS-DOS 2.11.22 operating system.
```

```
     15
CHECKING           SAVINGS            STOCKS & BONDS
FURN & APPLIANCES  TOOLS & EQUIPMENT  AUTOMOBILE
HOME AND LOT

                                      NOTE PAYABLE
MORTGAGE PAYABLE

                                      EQUITY
WAGE INCOME        INTEREST INC       DIVIDEND INC
GAIN ON SALES      SOC SEC INC

CAR EXPENSE        UTILITIES & PHONE  INSURANCE
REAL ESTATE TAX    REPAIRS            MEDICAL/DENTAL EXP
CONTRIBUTIONS      LOSS ON SALES      INTEREST EXPENSE

                                      DRAWING ACCOUNT
```

```
     1
 0 \ Nuts and bolts    Load screen                        10-19-88jm
 1 : 00    (S --)  0 0 ;
 2 : 4+    (S a --a)  4 + ;
 3 : D0<   (S d --f)  SWAP DROP 0< ;
 4 : D0>   (S d --f)  DNEGATE D0< ;
 5 : D0<>  (S d --f)  D0= NOT ;
 6 : 2+!   (S d a --) DUP >R 2@ D+ R> 2! ;
 7
 8 : (DC)  ." DEBIT      CREDIT " ;
 9 : (PL)  ." PROFIT AND LOSS   " ;
10 : (BS)  ." BALANCE SHEET     " ;
11 : (RC)  ." RECAP             " ;
12 : (PG)  ." POSTING             " ;
13 : (TB)  ." TRIAL BALANCE       " ;
14 : (FS)  ." FINANCIAL STATEMENT  " ;
15 2 14 THRU
```

```
     16
\ 1-16 Shadow -- Nuts and Bolts                             10-19-88jm

     Lines 1 thru 6 add useful words not in the F83
dictionary.


2+!  adds the double-length number on the stack to the amount
stored at the given address.

     Lines 8 thru 14 are headings used in various places for
clarity.


2 14 THRU      OK will now load the application source code.
```

```
     2
 0 \ Utilities                                            10-19-88jm
 1
 2 : .AC#  (S n--)  2 .R 2 SPACES ;  \ print account number
 3
 4 : .ACNAME  (S n--)            \ print account name
 5    15 BLOCK SWAP 1- DUP >R  (S addr ac#-1 --)
 6    20 * +                     \ adjust addr for 20-space names
 7    R> 3 / 4 * +               \ adjust addr for 64-space lines
 8    20 TYPE ;                  \ print the account name.
 9
10 : WHAT  (S a --a d d)         \ what is at this address?
11    DUP 2@ 2DUP ;
12
13
14
15
```

```
     17
\ 2-17 Shadow - Utilities                                   10-23-88jm

     Valid account numbers are 1 thru 48.
     When posting (see screen 8) an invalid account number
will exit the posting loop and grand total debits and credits.

     Account names are listed 3 to a line in screen 15. Please
note that AC#1 is CHECKING, #2 IS SAVINGS, #3 is STOCKS &
BONDS, #4 FURN & APPLIANCES and so on in that order across and
down.

     Numbers 1 thru 21 are reserved for ASSETS and LIABILITIES
             22 THRU 24        do      PROPRIETARY INTEREST
             25 THRU 48        do      INCOME and EXPENSES
```

3

```
\ Deferred words                                   10-20-88jm

DEFER ARRAY
48 4 * CONSTANT ARRAYSIZE
CREATE  (JRL) ARRAYSIZE ALLOT        \ general journal
CREATE  (LED) ARRAYSIZE ALLOT        \ general ledger
(LED) (JRL)  -  CONSTANT DIFFERENCE  \ for transfer purposes
: ACTADR  (S n--a)                   \ offset to account address
   1- 4 * ARRAY + ;
: JRL->LED  (S --)  \ Updates ledger with contents of journal
   (JRL) 48 0 DO DUP DUP 20 ROT DIFFERENCE + 2+! 4+
            LOOP DROP ;

DEFER BOOKS
: JL  ." JOURNAL        " ;
: LR  ." LEDGER         " ;
```

18

\ 3-18 Shadow -- Deferred words 10-20-88jm

The two identical arrays are the body of the system. The journal is used for current period data accumulation - daily, monthly or whatever, while the ledger is the year-to-date repository for that information.

ACTADR converts the account number to the address of the account balance.

JRL->LED is the lower level word that updates the ledger at the end of the current period.

The deferred word BOOKS is included in the headings for Posting, Trial Balance and Financial Statement to remind the user which of the two books is in current use.

4

```
\ Double-length number input/output               10-19-88jm

: INPUT  QUERY BL WORD NUMBER ;  (S --d)  \ stack a double-
                                          \ length number

: (D.)$  (S d--a l)  \ convert double-length number to a
   TUCK DABS         \ money string
   <# # # ASCII . HOLD #S ROT SIGN #> ;

: D.R$   (S d n--)   \ output a money string n spaces
   >R (D.)$ R> OVER - SPACES TYPE ;  \ right justified

: 12D.R$  12 D.R$ ;  (S d--)  \ output 12 spaces right justified
: 18D.R$  18 D.R$ ;  (S d--)  \    "   18   "
: 30D.R$  30 D.R$ ;  (S d--)  \    "   30   "
```

19

\ 4 -19 Shadow - Double-length number input/output 10-19-88jm

INPUT is the user interface for account number and amount.

The program responds to input with or without the decimal but for practical purposes account numbers should be entered without the decimal and all money amounts should be entered with the decimal in its proper place and including all zeroes.

Example: 25 dollars is entered as 25.00 (not 25.)

5

```
\ Debit/Credit utility                            10-19-88jm

2VARIABLE DEBITS   2VARIABLE CREDITS

: DC0  00 2DUP DEBITS 2!  CREDITS 2! ;  \ clear debits and
                                        \ credits to 00

: DEBIT?  (S d--d f)  2DUP D0> ;   \ is it a debit?

: .AMOUNT  (S d--)    \ print debit or credit amount
   DEBIT? IF 18D.R$ ELSE DABS 30D.R$ THEN ;
```

20

\ 5-20 Shadow - Debit/Credit utility 10-21-88jm

This utility manages the debit/credit input and output while the actual variables serve as accumulators.

If you're a little rusty in the double-entry area it helps to remember that for every debit there must be one or more credits and vice versa. Also, be it journal or ledger, for either book to be in balance the total of all debits must equal the total of all credits.

In this application debits are entered as positive values and credits as negative. The totaling process compares the absolute values.

```
      6
 \ Debit/Credit utility                              10-11-88jm

 : TOTALDCS  (S l i --)  \ total and store debits, credits
    DO I ACTADR 2@ DEBIT?
      IF DEBITS 2+! ELSE CREDITS 2+! THEN
    LOOP ;

 : .GTOTALS  (S --)  \ print grand totals debits, credits
    CR 49 1 TOTALDCS
    ."    TOTALS" 15 SPACES
    DEBITS 2@ 18D.R$ CREDITS 2@ DABS 12D.R$ DC0 ;
```

```
      21
 \ 6-21  Shadow -  Debit/Credit utility                  10-20-88jm

TOTALDCS  Scan a range of accounts.  Fetch and accumulate
     contents in the DEBIT and CREDIT accounts.

.GTOTALS  Scan all debits and credits in the current book.
     Fetch and accumulate contents in the DEBIT and CREDIT
     accounts.  Retrieve and print their total absolute
     values and clear the DEBIT and CREDIT accounts.
```

```
      7
 \ Posting utility                                   10-20-88jm

 : PGHEAD  (S --)  (PG)  BOOKS  (DC) ;    \ posting heading
 : ENTERAC#  (S --d)  ." ENTER ACT #  " INPUT ;
 : TESTAC#  (S d--n f)  DROP DUP 1 48 BETWEEN ;
 : WASH  (S --)  -LINE 13 EMIT ;   \ clears clutter
 : ENTERAMT  (S n--n d) DUP .ACNAME ." ENTER AMOUNT "
    INPUT WASH ;

 : ADDAMT  (S n d --d)        \ add to account
    ROT DUP >R   \ d n       save a copy of AC# on return stack
    .AC#         \ d         print account number
    R@ .ACNAME   \ d         print account name
    2DUP R>      \ d d n     prepare to add to account
    ACTADR 2+! ; \ d         make the addition
```

```
      22
 \ 7-22 Shadow  Posting utility                          10-21-88jm

     In this application all income is deposited in one
checking account and all outgo is disbursed by check from this
account.

     At regular intervals deposits and checks are posted via the
posting utility to the JOURNAL.  Entries that do not involve
the check book should also be made at this time.

     This process categorizes and summarizes the data.  At
the end of the posting session, when the debits and credits
are in balance, they are transferred (added) to the ledger.
```

```
      8
 \ Posting utility                                   10-20-88jm

 : PROCEED  (S f--f)     \ proceed with entries
    IF ENTERAMT ADDAMT .AMOUNT TRUE
    ELSE DROP FALSE
    THEN ;

 : CONTINUE  (S --) CR PGHEAD  \ continue posting
    BEGIN CR ENTERAC# TESTAC#
      PROCEED
    WHILE
    REPEAT .GTOTALS ;
```

```
      23
 \ 8-23 Shadow  Posting utility                          10-20-88jm

CONTINUE  is the lower level word that sets up the posting
     process between user and computer.

     It requests data in the form of account number and
     amount until the user enters an account number other
     than 1 to 48 at which time it exits the loop,
     totals the debits and credits and displays the totals for
     comparison.  To re-enter the loop use POST.
```

```
9
\ Trial balance                                    10-13-88jm
: TBHEAD  (S --)  (TB) BOOKS (DC) ;  \ trial balance heading

: LISTACTS  (S addr limit index --)  \ list certain active acts
   DO WHAT D0<>
     IF I DUP .AC# .ACNAME .AMOUNT CR
     ELSE 2DROP
     THEN 4+
   LOOP DROP ;

: TRIAL-BALANCE  (S --)  CR TBHEAD   \ trial balance
   CR 1 ACTADR 49 1 LISTACTS .GTOTALS ;
```

```
24
\ 9-24 shadow  Trial Balance                       10-22-88jm
```

TRIAL-BALANCE is the lower level word that examines the entire contents of either book at any time.

It is particularly useful during the posting session because one can see the effect of any and all entries simply by alternating between the posting loop and the trial balance.

```
10
\ Financial statement                              10-14-88jm
: FSHEAD   (S --)  (FS) BOOKS (PL) ;  \ main statement heading
: FSHEAD1  (S --)  21 SPACES BOOKS (BS) ;  \ 1st subhead
: FSHEAD2  (S --)  21 SPACES BOOKS (RC) ;  \ 2nd subhead

: PRINTENTRY  (S a d n n --a)   \ print ac#, acname and amount
   .AC# .ACNAME DABS 10D.R$ ;


: LISTDEBITS    (S address limit index --)  \ list debits only
   DO WHAT D0> IF CR I DUP PRINTENTRY ELSE 2DROP THEN 4+
   LOOP DROP ;

: LISTCREDITS   (S a l i --)     \ list credits only
   DO WHAT D0< IF CR I DUP PRINTENTRY ELSE 2DROP THEN 4+
   LOOP DROP ;
```

```
25
\ 10-25 shadow  Financial Statement                10-23-88jm
```

The financial statement includes:

1. A profit and loss section
   Income minus Expenses = Net Profit or Loss
2. A balance sheet
   Assets minus Liabilities = Net Worth or Deficit
3. A recapitulation of Net Worth and Owner's Equity
   Owner's Equity at start of period
   plus or minus profit or loss =
   Owner's Equity at end of period = Net Worth or Deficit

Financial statement format is different from the trial balance in that debits and credits no longer have separate columns and negative values are introduced for a net loss &/or deficit.

```
   11
 0 \ Financial statement                              10-15-88jm
 1 : ASSETS         ." ASSETS     " 1 ACTADR 22 1 LISTDEBITS ;
 2 : LIABILITIES    ." LIABILITIES" 1 ACTADR 22 1 LISTCREDITS ;
 3 : INCOME   ." INCOME " 25 ACTADR 49 25 LISTCREDITS ;
 4 : EXPENSE  ." EXPENSE" 25 ACTADR 49 25 LISTDEBITS ;
 5
 6 CREATE (P&L) 0 , 0 ,  \ transitory profit and loss account
 7
 8 : P&L  (S --)    FSHEAD   CR        \ profit and loss
 9    49 25 TOTALDCS
10      INCOME   CREDITS 2@ 2DUP DABS 12D.R$ CR
11      EXPENSE  DEBITS  2@ 2DUP      12D.R$ CR
12        D+ 2DUP (P&L) 2! DNEGATE
13      ."   NET GAIN (LOSS -)" 17 SPACES 10D.R$ DC0 ;
14
15
```

```
26
\ 11-26 shadow  Financial Statement                10-23-88jm
```

Profit and Loss

line 8 Print the statement heading.
line 9 Total and store debits and credits included in the income/expense section AC#'s 25 to 48.

line 10 List the credits; fetch, duplicate and print the total.
line 11 List the debits; fetch, duplicate and print the total.

line 12 Add the credits to the debits on the stack.
Duplicate the result.
Store one copy in the transitory (P&L).
Change sign of the copy on the stack.

line 13 Print the result as net gain or loss.

12

```
\ Financial Statement                               10-25-88jm
: BAL  (S --)   FSHEAD1     CR           \ balance sheet
   22 1 TOTALDCS
     ASSETS        DEBITS  2@ 2DUP      12D.R$ CR
     LIABILITIES   CREDITS 2@ 2DUP DABS 12D.R$ CR
       D+
     ."   NET WORTH (DEFICIT -)" 13 SPACES 18D.R$ DC0 ;

: RECAP  (S --)  FSHEAD2        \ recap net worth and equity
     24 ACTADR 2@ 2DUP DNEGATE
   CR ." OPENING EQUITY" 10 SPACES 12D.R$
     (P&L) 2@ 2DUP DNEGATE
   CR ."   NET GAIN (LOSS -)" 5 SPACES 12D.R$
     D+ DNEGATE
   CR ." CLOSING EQUITY (DEFICIT -)" 10 SPACES 18D.R$ DC0 ;
```

27

```
\ 12-27 shadow  Financial Statement                 10-25-88jm
Print the balance sheet subhead.
Total & store asset/liability DC's -- AC#'s 1 thru 21.
List the debits; fetch, duplicate and print the total.
  "    "  credits;  "      "      "      "     "     "
Add the debits to the credits on the stack.
Print the resulting net worth or deficit.

Print the recap subhead.
Fetch opening equity from AC# 24, duplicate & change sign.
Print it.
Fetch profit or loss, duplicate and change sign.
Print it.
Add the amounts on the stack and change sign.
Print the result and clear debits and credits.
```

13

```
\ High level words                                  10-20-88jm

: JOURNAL  ['] (JRL) IS ARRAY ['] JL IS BOOKS ; \ activate jrnal
: LEDGER   ['] (LED) IS ARRAY ['] LR IS BOOKS ; \ activate ledgr

: CLEAR-JOURNAL  (S --)  (JRL) ARRAYSIZE ERASE ;
: CLEAR-LEDGER   (S --)  (LED) ARRAYSIZE ERASE ;

: TB  (S --)  TRIAL-BALANCE ;            \ trial balance

: FS  (S --)  CR P&L CR BAL CR RECAP ;   \ financial statement
```

28

```
\ 13-28 shadow  High level words                    10-25-88jm

Make JOURNAL current for posting, trial bal and fin stmt.
  "   LEDGER     "     "    "        "    "    "   "   "

Clear all JOURNAL accounts to zero.
  "    "  LEDGER    "     "  "

Print the contents of the current book in trial balance
     form.
Print the contents of the current book in financial
     statement form.
```

14

```
\ High level words                                  10-25-88jm

: NEWBOOKS  (S --)  \ begin journal/ledger from all acts zero
    CLEAR-JOURNAL CLEAR-LEDGER JOURNAL CONTINUE ;

: POST  (S --)  CONTINUE ;  \ continue posting current book


: TRANSFER  (S --)  JRL->LED  CLEAR-JOURNAL ;  \ transfer data
                                               \ to ledger --
                                               \ clear journal
```

29

```
\ 14-29  shadow High level words                    10-27-88jm

NEWBOOKS   Just post your assets and liabilities to the
           appropriate accounts, check the totals and enter
           the difference as your equity in account #24 and
           you're in business!

POST       post is the workhorse command that receives original
           entry data and enters it in the current book.

TRANSFER   adds the contents of the journal to the ledger and
           clears the journal for the next period's entries.
```

Forth-83

DEVELOPING A STEP TRACE

CHESTER H. PAGE - SILVER SPRING, MARYLAND

It is convenient to have a STEP-TRACE routine which displays the parameter stack (and the floating-point stack, if appropriate), the name of the word being executed, and the resulting stack(s). I have developed such a routine with some interesting features, and a development technique involving three stages.

The first stage makes brute-force use of high-level variables and constants, and a Forth assembler. The second stage is a little more elegant: most of the intermediate parameters are replaced by dummy numbers and addresses. These are overwritten at the end of the assembly, using location data about the words just defined. The basic reason for these maneuvers is that there is a circular dependence of definitions upon each other, so no order of defining the words allows for a simple succession of definitions. For example, DETOUR uses (UNDETOUR), which uses (DETOUR), which uses DETOUR.

The final version provides a more elegant stack display.

In both these stages, an assembler must be loaded and used. It is more convenient to have definitions that can be added to a dictionary by a simple screen loading; the third stage provides this. It is achieved by developing the primitive words in stage two, and providing for defining these by compiling bytes, using CREATE. The final version provides a more elegant stack display (aligned four-digit hex numbers) and al-

```
TRACE SCR # 1
 0 \ Preliminaries                                        30JUL88CHP
 1
 2 \ Boot FORTH
 3 \ Define : DUMMY ;
 4 \ Enter HEX 2000 ALLOT
 5 \ Load ASSEMBLER
 6 \ Load TRACE
 7
 8 \ This manouver combined with Screen 2, line 5 and Scr 5, L 7
 9 \ eliminates ASSEMBLER and the temporary constants of Scr 2,
10 \ L 2/3, from the final dictionary
11
12 -->
13
14
15


TRACE SCR # 2
 0 \ Parameters and stack print                           30JUL88CHP
 1 HEX
 2 EE CONSTANT IP
 3 F1 CONSTANT W
 4
 5 ' DUMMY 4 + DP !
 6
 7 VARIABLE FLOOR
 8 VARIABLE FROM
 9 VARIABLE TEMP
10
11 : .S DEPTH ?DUP IF 0 DO DEPTH I - 1- PICK . LOOP
12     ELSE ."  Empty stack" THEN ;
13
14 -->
15


TRACE SCR # 3
 0 \ (UNDETOUR), DETOUR                                   30JUL88CHP
 1 ASSEMBLE (UNDETOUR) PLA, IP STA, PLA, IP 1+ STA,
 2     PLA, W 1+ STA, PLA, W STA,
 3 \ Reset detour
 4     TEMP 1+ LDA, ' NEXT 1A + STA,
 5     TEMP LDA, ' NEXT 19 + STA,
 6 \ Proceed with original word
 7     0 # LDY, W 1- JMP,
 8
 9 : DETOUR >R .S KEY DROP R> CR >NAME ID. 4 SPACES (UNDETOUR) ;
10
11 -->
12
13    TEMP is a substitute for the Parameter Field Address
14      of (DETOUR) to break a circular dependence.
15
```

lows reverting to normal operation even during a trace.

Operating Principles

Entering `TRACE` enables a detour signpost `(DETOUR)`, a jump to which is substituted for the `JMP W-1` at the end of `NEXT`. If the word request (i.e., the parameter field entry containing the code field address of the requested word) is below a specified `FLOOR`, the detour is ignored. This avoids having components of components of components analyzed *ad nauseum*. `FLOOR` defaults to the original dictionary top, but can be moved down to allow tracing words defined before `TRACE` was added.

When the detour is taken, the code field address of the word to be executed is put on the parameter stack for use in printing its name, and on the return stack for storage. The "detour sign" is then removed (for the sake of later arrivals) and the detour is taken. While in the detour, the parameter stack is printed (and the floating-point stack, if desired). `DETOUR` is a colon word, so variations are easily added. The last component of `DETOUR` is the primitive `(UNDETOUR)`, which recovers `IP` and `W` (the interpretive pointer pointing at requests, and the word pointer), resets the detour sign, and proceeds with the original command via `JMP W-1`, as was intended at the end of `NEXT`.

Since `DETOUR` is a colon word, the `IP` that called it was put on the return stack, to be recovered by `EXIT` (called by the semicolon); but the last component of `DETOUR` is a primitive that ends in a `JMP` command, so that the semicolon is never reached! To replace its action, `(UNDETOUR)` must start by pulling the stored `IP` off the return stack, and storing it in the `IP` pointer.

Having `(DETOUR)` remove the detour signpost before taking the detour protects `DETOUR` itself from being traced, avoiding an infinite loop of self-tracing. By restoring the signpost *after* the detour is finished, the next word external to the detour operation will be traced.

Interrupting the detour with `KEY` provides for tracing one step at a time for each press of the spacebar; holding the spacebar down provides continuous tracing. Pressing `<DELETE>` aborts the operation; any other key continues the operation in normal mode (no trace). When the trace of a word is finished, the routine awaits the

(Text and screens continued on page 27.)

```
TRACE SCR # 4
\ (DETOUR)                                            30JUL88CHP
ASSEMBLE (DETOUR)
   SEC, IP LDA, 2 # SBC, FROM STA, IP 1+ LDA, 0 # SBC,
   FLOOR 1+ CMP, 101 BCC, 102 BNE, FROM LDA, FLOOR CMP,
   101 BCC,
   102 DEX, DEX, W LDA, PHA, 0 ,X STA,
            W 1+ LDA, PHA, 1 ,X STA,
\ Directions for detour
   103 ' DETOUR 100 /MOD # LDA, W 1+ STA,
                         # LDA, W STA,
\ Remove detour signpost
   F0 # LDA, ' NEXT 19 + STA, 0 # LDA, ' NEXT 1A + STA,
   101 0 # LDY, W 1- JMP, END   -->
END sets the branches to the labels 101, 102, etc.
103 is a dummy label; /MOD puts two numbers on the stack,
the first would be misinterpreted as a label if no label
```

```
TRACE SCR # 5
\ TRACE, NOTRACE, Relink dictionary                   30JUL88CHP

: TRACE ['] (DETOUR) 2+ ['] NEXT 19 + ! ;
: NOTRACE F0 ['] NEXT 19 + ! ;

' (DETOUR) 2+ TEMP !

' DUMMY >NAME ' FLOOR >LINK !
\ Establishes a link bypassing the assembler

HERE FLOOR !

QUIT
```

```
TRACE SCR # 6
\ Second stage of development                         30JUL88CHP
HEX
EE CONSTANT IP
F1 CONSTANT W
F4 CONSTANT FROM

' DUMMY 4 + DP !

VARIABLE FLOOR

: .S DEPTH ?DUP IF 0 DO DEPTH I - 1- PICK . LOOP
     ELSE ."  Empty stack" THEN ;

-->
```

```
TRACE SCR # 7
\ Second stage, continued                             30JUL88CHP
ASSEMBLE (UNDETOUR) PLA, IP STA, PLA, IP 1+ STA,
     PLA, W 1+ STA, PLA, W STA,
\ Reset detour
     FF # LDA, ' NEXT 1A + STA,
     FF # LDA, ' NEXT 19 + STA,
\ Proceed with original word
   0 # LDY, W 1- JMP,

: DETOUR >R .S KEY DROP R> CR >NAME ID. 4 SPACES (UNDETOUR) ;

-->
```

MULTITASKING & CONTROLLING REGULAR EVENTS

T. HENDTLASS - HAWTHORN, AUSTRALIA

One of the requirements of real life is to perform multiple tasks at regular intervals. Forth does not provide this real-time capability directly; it can perform multiple tasks apparently simultaneously by using multitasking, but it has no internal timer to schedule events at specified times. With such a timer per task, and with the multitasker, we can arrange for events to occur at predestined times, or at least very close to them. This paper describes a timer for use with the IBM PC family, and discusses the multitasker built into the F83 public-domain Forth system.

The virtues of simplicity are nowhere stronger than in multitasking.

Figure One. The definition of the defining word `TIMER`.

```
: TIMER
  CREATE ( -- )          \ no stack effect when creating
    4 ALLOT              \ space for two variables
  DOES>  ( -- adr )      \ run-time stack effect of creation
     (READ_CLOCK)        \ get new_value from clock
     OVER 2+ @           \ and last value
     OVER -              \ calculate change
     2 PICK +!           \ update user value
     OVER 2+ !           \ save latest value read
  ;
```

Figure Two. A version of `(READ_CLOCK)` for F83 on a PC.

```
code (READ_CLOCK)     ( -- n )
0 # mov                \ Ah=0 to read clock
26 int                 \ 1Ah=26 is the real-time clock
dx ax mov              \ low 16 bits of answer to ax
1push                  \ answer to stack and exit to next
end-code
```

Figure Three. F83 provides these multitasker-interface words.

`SINGLE ( -- )`
Disable multitasking by vectoring `PAUSE` to a null word. Leave the current task running as the only task, but don't alter the circular linked list of tasks.

`MULTI ( -- )`
Enable multitasking by vectoring `PAUSE` to the active word `(PAUSE)`, which handles the task interchange.

`BACKGROUND: ( -- )`
Contains a defining word that defines a task in the round-robin multitasker. It allocates a stack area of 400 bytes (100 for the return stack and 300 for the data stack) and links the task, leaving it in the sleeping condition. Typing the task name will return its address, rather than activating it; it can only be run by the multitasker. See comment on this name, in text.

`WAKE ( adr -- )`
Wake up the task whose address is on the stack, so that it will execute in its next turn.

(Continued.)

Of Tasks and Timers

First, each timer is set to an initial value. Every task checks its timer whenever the multitasker runs it. If time is up, it does whatever needs to be done and resets the timer to its initial value; if not, it just passes control onto the next task. The accuracy of the timing depends on the frequency of the task interchange in the multitasker and on the resolution of the timers. The rate of task interchange is under the control of the programmer: a task exchange takes place whenever the word `PAUSE` is executed. Although it can be placed liberally throughout the code and every input or output word has `PAUSE` embedded in it, this is the major cause of latency and the timer need not have a very high resolution. For tasks that have to run at, say, intervals of minutes, it is not hard to arrange things so that the maximum time latency is only on the order of a second or so.

All we need to add to standard Forth are the timers. One method of achieving this is with a new defining word which I have called TIMER. This creates a timer which can be preset to a value and which will be decremented at a known rate. Periodic checking of the value in this timer will provide the cue to run the task associated with this timer. Although only one new word, TIMER, is added for direct use, the system-dependent part of the definition is factored into another word called (READ_CLOCK). When called, (READ_CLOCK) leaves a number on the top of the stack; this number must be maintained by the host computer hardware in some way, increasing at a regular and known rate. In the IBM PC family, a suitable timer is available and may be obtained by reading the DOS real-time clock.

(Continued.)

```
SLEEP   ( adr -- )
```
Make the addressed task pause indefinitely until it is woken again (if ever).

```
STOP   ( -- )
```
Put the current task to sleep. If a task ends (i.e., doesn't run continuously in an endless loop), then it must end with this word. Otherwise, a task will try to execute its stacks with unpredictable—but certainly very undesirable—results.

```
PAUSE   ( -- )
```
The task in which this word appears stops, and control is passed to the next task in the list. PAUSE exists in all input and output words except those directly involving input and output ports. If none of these words are used (implicitly or explicitly), the task will never release control to the next task.

```
ACTIVATE   ( -- )
```
Force the assigned task to execute new code rather than its old code.

Figure Four. Example use of F83's multitasking words.

```
BACKGROUND:  PRINT*S
  20 0 DO                          \ set up outer loop
    ASCII * EMIT                   \ send one *
    100 0 DO  PAUSE LOOP           \ wait a bit
  LOOP                             \ loop to send next
  STOP  ;
```

Figure Five. A 'multitasker-safe' version of the previous example.

```
: NEW-PRINT*S
  PRINT*S ACTIVATE
  BEGIN                              \ set up outer loop
    20 0 DO                          \ set up inner loop
      ASCII * EMIT                   \ send one asterisk
      100 0 DO  PAUSE LOOP           \ wait a bit
    LOOP                             \ loop to send next
    STOP FALSE                       \ stop when 20 sent
  UNTIL  ;                           \ loop forever
```

Figure Six. The formal definition of BACKGROUND:.

```
: BACKGROUND:
  400 TASK:    \ define a task entry with 400 bytes
               \ for the stack, and the name following
  HERE         \ pointer to where code will be compiled
  @LINK 2-     \ address of task just defined
  SET-TASK     \ initialize the new task
  !CSP         \ initialize compiler error checking
  ]            \ compile the code the follows, so
               \ it will be executed by this new task
  ;
```

An example use of TIMER is:

```
TIMER name
```

which creates a timer called name.

Name, when run, returns the address where the count for this timer is held, so that it can be initialized with a normal store or can be read with a normal fetch. However, name does more than that. When it is called, it updates the value in its counter (based on the amount of time since it was last updated) before it returns the counter address. This updating is done on a when-needed basis to save processing time, as the value in the counter need not be updated until it is to be read (obviously) or initialized (less obviously).

Internally, each timer keeps two values: the user initializes and reads the *user value*, which steadily counts down from the initial value to zero (and beyond!); the *internal value* is the value obtained from the system clock the last time it was read. When a timer is activated, it reads the system clock and subtracts the previous system clock value (obtained from the internal value). Then it decreases the user value by this amount and updates the internal value. When a timer is being initialized, both the user and internal values need to be set, otherwise the first read of the timer will produce unpredictable results.

Defining a Defining Word

The new defining word TIMER is itself defined with the words CREATE and DOES>. For those not familiar with the operation of CREATE and DOES>, a brief explanation follows.

A defining word has two quite distinct parts: one describes what the defining word is to build, and the other consists of the behavioral characteristics of the new entity it builds. For example, consider the processing of:

(Continued on page 30.)

Forth-83

BINARY TABLE SEARCH

DAVID ARNOLD - KIRKSVILLE, MISSOURI

A binary search of a table can be remarkably quick and can be adapted readily to various types of data. The table records must be arranged in order, and none may be duplicated. The search starts by declaring the whole table as a search region. Then a test datum is compared with a record near the middle of the region. If they match, the search ends. Otherwise, another midpoint test is made. If the test item was larger than the inspected table item, the upper part of the current search region becomes the next search region. If the test data was smaller, the lower part of the current region is searched next. If a table record exists that can match the test data, the search homes in on it. Otherwise, the table is soon exhausted, and the search ends unsuccessfully.

Usually, a part of each record called the key field is set aside for a datum of a type that can be easily ordered and compared, and which can be used as a label for one and only one record in the table. The key fields may contain useful information, or they may be used just to make it convenient to order and search the table. Other fields in the record may hold information that isn't easy to put in order or to compare, or that may be duplicated or blank in some records. For example, a voter registration list might list one voter in each record. A three-field record could hold a voter's name, home address, and social security number. The name and the address could be stored in two text fields, and the social security number in a numeric field. The social security number field would make a good key field: Numbers are easier to order and compare than text and, barring errors, no two people are assigned the same number. Though the name be misspelled and the address wrong or absent, the number could still be used to locate the record.

`BIN_SRCH` does a binary table search. It receives three items on the stack, 1) the address of a table, with its records arranged so their key fields are ordered small to large and no key fields are duplicated; 2) the number of records in the table; and 3) a test datum which is tested for a possible match with some key field in the table. If a match is found, the address of the matching record is returned on the stack. If none was found, a false flag is returned. [1]

An average successful search requires log2(N)-1 comparisons.

There are two possible exit points. If a match is found, it immediately returns; otherwise, it eventually exhausts the table, exits the search loop, and returns. If it starts with a table of zero length, execution falls through to the code that returns a false flag, as if an unsuccessful search had been done.

To start the search, the whole table is defined as the current search region. Two variables on the stack hold the lower and upper table indices of the current region. During each pass through the search loop, the key field in a record at the middle of the region is compared with the test data. If the two match, the address of the just-inspected record is left on the stack and the word returns. Otherwise, a new search region is defined. If the test data was greater than the contents of the key field, the index of the record following the one just tested becomes the new lower bound. If the test data was the smaller, then the index of the record preceding the one just tested becomes the new upper bound. Then a new pass through the search loop tests another middle record. [2] If no match exists, the lower and upper bounds eventually cross each other, and the putative upper index is less than the lower. The loop termination test finds this and exits the loop. At that point, a false flag is left on the stack, and the word returns.

`BIN_SRCH` uses some Forth-83 double-number operators to manipulate *pairs of stack variables*, not double-precision numbers. If you're using a 32-bit system, you might want to check these words to be sure they work with a pair of stack items, not just with one natural, double-precision-sized machine word. [3]

`1 LOAD` will load everything. `ONLY FORTH DEFINITIONS ALSO` sets up the search order. [4] Laxen and Perry's F83 sets the search order thus. On systems such as fig-FORTH that set up the search by linking vocabularies when they're compiled, `FORTH DEFINITIONS` would do.

Screen three contains words that handle the table records. Redefinition of these words would allow access and comparison of various types of records and the data therein.

Screens five through seven contain words to demonstrate table searching. `KEY>FUNC` receives the address of a table of records, and a test keycode. The first item in the table is the number of records. After that, the records are listed. Each record holds a keycode in the first field and a function address in the second. If a keycode match is found, a corresponding function address is returned; otherwise, it returns a false flag. `KEY_DEMO` uses `KEY>FUNC` to search some sample keycode/function tables. The sample functions just print a few things on the console display. If no table record matches the test keycode sent to `KEY>FUNC`, you get beeped at.

How fast is this binary search? If `N` is the number of table records, and 2^K is the smallest power of two that is larger than N, then the greatest number of comparisons needed to exclude a match is K. A successful search could take as many as K comparisons. The average number of comparisons for a successful search would be about log2(N)-1. [5] (Log2(x) is the logarithm to the base two, and is equal to ln(x)/ln(2).) For example, searching a table of 25 keycode/function records, suitable for `KEY>FUNC`, would take no more than five comparisons—since 32 (2^5) is the smallest power of two greater than the table size—and the average number of comparisons during successful table searches would be about log2(25)-1 ≈ 3.6.

Screens eight through nine contain some words to set up a test table and run some speed tests. On my 7 MHz IBM PC-compatible computer, with the non-Forth-83 Standard words defined in high-level Forth, the time to set up and call `KEY>FUNC` averages between seven and nine milliseconds per search of a 256-element table. Generally, the greater the likelihood of finding a match, the less time a search takes.

Constraints and Possibilities

`BIN_SRCH` must use table indices instead of absolute addresses to specify its search region—even with tables of simple data like characters or integers—because the operation that finds the middle element does so by averaging the regions' limits, and the intermediate sum of the two addresses might exceed Forth-83's range of 16-bit unsigned integers (i.e., 65535). And to swiftly divide that sum, `2/` is used; it does signed division, and the sum of the two addresses might exceed the range of positive signed integers (i.e., 32767). [6]

The tables delivered to `BIN_SRCH` must have no more than 16383 records. That keeps the intermediate sum of the index limits within the range of positive signed integers. A big integer array for a small program could be larger than that—even in a 16-bit address space—and a virtual array in disk storage could be huge. With modified table-access words, indices in the range -16383 to 16383 might be used, doubling the workable table size. With a modest loss of speed, `D+` and `UM/MOD` might be used to average the search region's limits, and `D<` could be used for the test at the end of the search loop.

Other search methods that also progressively approach a matching table record are described in the book by Knuth and seem well suited to Forth. A binary search that specifies a search region and center record not with three variables (the upper, lower, and center indices) but with two (the center index and its distance from the center of the region last checked) might be a bit faster, and could use indices in the range zero to 32767. A search that uses Fibonacci numbers needs only the speedy addition and subtraction operations to locate the next record to test, and would not have oversized intermediate results. A table whose records contain pointers [7] that explicitly trace out branching relationships among the data in the records can have records deleted and inserted without requiring that the rest of the table be shifted around.

Notes

[1] A valid address must be non-zero, and a false flag is the quantity zero.

[2] The search paths trace out the branches of a tree-like pattern. Each middle record corresponds to a fork (called a node) in the tree. The leftward branch (if one exists) and all its subsequent nodes would hold data that is less than the aforementioned fork; and a rightward branch and all its nodes would be greater. In a plain ordered table, the algorithm implicitly describes a binary tree.

[3] Forth-83, the latest codification of conventional Forth practice, specifies that single-precision numbers be 16 bits long, the word size used by most Forth words. `DUP`, `SWAP`, and `ROT` are some prominent examples. 32-bit double-precision numbers are handled as pairs of single-precision numbers, and a set of double-number operators such as `2DUP`, `2SWAP`, etc. are generally used on those longer numbers. The double-number operators are also useful for working with pairs of numbers on the stack, when the word size is less of an issue than the fact that the numbers are distinct, not components of a double-precision

number. For example, 2SWAP is tidier than ROT >R ROT >R, and if it's available in machine code, it is faster. It happens that 16-bit words are the size most conveniently handled by the most common small computers, and Forth systems running on them often have double-number operators. Some of the newer (and more expensive) small machines can handily work on 32-bit numbers, and it would be possible for a Forth system running on them to omit double-number operators and make do with the machine's natural ability to use double-precision numbers. I have never used such a computer, though, and can't say how likely that would be.

[4] This is an experimental proposal by William F. Ragsdale (*Forth-83 Standard*, pp. 61–65). CONTEXT is an array of vocabulary addresses. When a word must be found in the dictionary, the listed vocabularies are searched in order, starting with the first array item. CURRENT is a variable that holds the address of the compilation vocabulary, into which words are to be compiled. A vocabulary, when executed, puts its address into the first location in the CONTEXT array, replacing whatever was there before. DEFINITIONS copies the first item of the CONTEXT array into CURRENT. ONLY is a vocabulary with special actions. It clears the CONTEXT array and puts its address in the first and last array locations. The ONLY vocabulary contains a few words that provide access to the other regular vocabularies. ALSO shifts all the CONTEXT items (except the ONLY item at the end of the array) one position toward the end of the list and leaves the leading item duplicated. The second ONLY item at the end of the array is not disturbed. Thus,

```
ONLY 1ST ALSO 2ND DEFINITIONS
ALSO
```

would make the search order:

```
2ND 2ND 1ST ONLY
```

and 2ND would be the compilation vocabulary. Additionally, ALSO is often used to leave the first item duplicated, because compilation of a colon definition starts by putting the contents of CURRENT into the first location of CONTEXT.

[5] *The Art of Computer Programming* (Vol. 3, 2nd ed.) by D.E. Knuth, has a technical description of this and other binary search methods. That fairly readable, unpatronizing seven-volume tome is chock-full of practical data processing methods. It might be available from a nearby college library, or a small public library might obtain it though an interlibrary loan.

[6] On Forth-83 systems, 2/ produces a floored quotient, corresponding with the floored results of Forth-83 division. The remainder has the sign of the divisor. Operations such as 2/, /, or MOD, which don't produce both quotient and remainder, produce results as if /MOD SWAP DROP or /MOD DROP had been performed. If you have a Forth-83 Standard system, try these division operators on some negative numbers. If floored division still seems mysterious, try multiplying the divisor and the floored quotient, then add that product to the floored remainder; the result should be the dividend. I've seen one Forth system that incorrectly implemented 2/ as a

```
Scr # 1     Forth-83
( binary table search                                     11Apr89dna )
ONLY  FORTH  DEFINITIONS  ALSO   DECIMAL
CREATE BINSRCH_MARK                           ( FORGET'able marker )
: ?ENOUGH  ( n -- )
 1+ DEPTH SWAP  U< ABORT"  ? Not enough parameters." ;
: NTHRU  ( start end -- )
 2 ?ENOUGH   OVER OVER 1+
 U< IF   1+  SWAP  DO   I U.  I LOAD   LOOP   THEN ;

 2 9  NTHRU
ONLY  FORTH  DEFINITIONS  ALSO   DECIMAL


Scr # 2     Forth-83
( stack ops  system specifics  miscellany                 16Mar89dna )
: 2DUP  ( n1 n2 -- n1 n2 n1 n2 )  OVER OVER ;
: 2DROP ( n n -- )  DROP DROP ;
: 2SWAP ( n1 n2 n3 n4 -- n3 n4 n1 n2 )  ROT >R  ROT R> ;
: 2OVER ( n1 n2 n3 n4 -- n1 n2 n3 n4 n1 n2 )  3 PICK  3 PICK ;
: NIP   ( n1 n2 -- n2 )  SWAP DROP ;
: -ROT  ( n1 n2 n3 -- n3 n1 n2 )  ROT ROT ;
: 2*    ( n -- 2*n )  DUP + ;
: U*    ( u1 u2 -- u3 )  UM* DROP ;
: NSGN  ( n -- -1 | 0 | 1 )  DUP IF   0> 2 AND  1-   THEN ;
0   CONSTANT FALSE
-1  CONSTANT TRUE                  ( 111..111B )
2   CONSTANT /N                    ( size of natural machine word )
      ( 2 bytes for 16-bit system   4 bytes for 32-bit system )
: BEEP  ( -- )  7 EMIT ;
: HEX   ( -- )  16 BASE ! ;

Scr # 3     Forth-83
( table access & comparison                               17Mar89dna )
2 /N *  CONSTANT /T                         ( size of table record )
: /T*   ( #tabl_rcrd -- size )  /T U* ;
: TA+   ( tabl_base idx -- tabl_rcrd_adr )  /T* + ;
: T>K   ( tabl_rcrd_adr -- rcrd_key_adr ) ;   IMMEDIATE
: T>F   ( tabl_rcrd_adr -- rcrd_cfa_adr )  /N + ;
: TCOMP ( n1 n2 -- -1 | 0 | 1   < = > )  - NSGN ;
```

simple bit shift, thus performing unsigned division, so you might want to check for that too.

[7] A pointer is a variable that contains an address. In this example, each table record would contain one or more pointers that each hold the address of the next record up or down in the branching pattern.

David Arnold was attracted to Forth because it compiles fast code and because a programmer can extend and refine it from the system roots up. He started with a Commodore 64, then got F83, and wound up writing a system from scratch to run on a PC clone. A disabled person, he is working toward earning a living in a restricted environment.

```
Scr # 4     Forth-83
( binary search for a matching record                    17Mar89dna )
: BIN_SRCH  ( tabl_adr tabl_siz srch_key -- rcrd_adr ! f= )
 OVER IF                                        ( table not empty? )
   >R   1- 0           ( .. -- tabl1 high_idx low_idx    -r- n )
   BEGIN
     2DUP  + 2/  3 PICK OVER TA+ ( .. -- tb h l m rc -r- n )
     DUP T>K @ R@  TCOMP          ( .. -- tb h l m rc ?  -r- n )
     ?DUP 0= IF
       R> DROP  >R  2DROP 2DROP  R>  EXIT         ( -- rcrd_adr )
     THEN                         ( .. -- tb h l m rc ?  -r- n )
     NIP   0< IF  1+  ELSE  1- -ROT  THEN   NIP
   2DUP < UNTIL                   ( .. -- tabl hghi lowi  -r- n )
   R> DROP
 THEN                                              ( .. -- x x x )
 2DROP DROP   FALSE ;                              ( .. -- f= )

Scr # 5     Forth-83
( search keycode/function tables  sample functions  18Mar89dna )
: KEY>FUNC  ( ktabl key -- cfa ! f= )
 OVER /N +  ROT @  ROT  BIN_SRCH   DUP IF  T>F @  THEN ;

: SHOW_LOW  ( c -- )
 CR  ." '^"  DUP 96 + EMIT  ." '"  2 SPACES  U. ;
: SHOW_CHR  ( c -- )
 CR  ." '"  DUP 32 MAX EMIT  ." '"  3 SPACES  U. ;
: SHOW_SPC  ( x -- )  CR  DROP  ." 'spc'"  SPACE  32 U. ;
: KEY_QUIT  ( x -- )  CR  DROP  ." 'quit_demo'"   CR  QUIT ;

Scr # 6     Forth-83
HEX ( keycode/function tables                            26Feb89dna )
CREATE LOWKEYS  0 ,  ( # keycode/function entries )
  1 ,  ' SHOW_LOW ,   2 ,  ' SHOW_LOW ,                  ( ^A ^B )
  3 ,  ' SHOW_LOW ,  1B ,  ' KEY_QUIT ,                  ( ^C esc)
HERE  LOWKEYS /N +  -  /T /  LOWKEYS !

CREATE HIGHKEYS  0 ,
 20 ,  ' SHOW_SPC ,  41 ,  ' SHOW_CHR ,                  ( spc A )
 42 ,  ' SHOW_CHR ,  43 ,  ' SHOW_CHR ,                  ( B C )
 61 ,  ' SHOW_CHR ,  62 ,  ' SHOW_CHR ,                  ( a b )
 63 ,  ' SHOW_CHR ,                                      ( c )
HERE  HIGHKEYS /N +  -  /T /  HIGHKEYS !
DECIMAL
```

```
Scr # 7     Forth-83
( select kybd functions                              26Feb89dna )
: ?DO_KEY  ( c cfa ¦ x f= -- )
 ?DUP IF   EXECUTE   ELSE   DROP  BEEP   THEN ;

: KEY_DEMO  ( --  )
 CR  ."  Key_demo  Press ESC to quit. "
 BEGIN
   KEY
   DUP 32 U< IF  LOWKEYS  ELSE  HIGHKEYS  THEN ( .. -- key tab )
   OVER KEY>FUNC                          ( .. -- key cfa ¦ x f= )
   ?DO_KEY
 0 UNTIL ;
```

```
Scr # 8     Forth-83
( make & fill test table                             11Apr89dna )
CREATE TEST_MARK                          ( FORGET'able marker )
CREATE TEST_TABLE   256 /T*  /N +  ALLOT
: FILL_TABLE  ( n_step -- )
 1 ?ENOUGH   0 TEST_TABLE !
 256 0 DO
   TEST_TABLE /N +  I TA+   OVER I U*     ( .. -- nstep rcrd n )
   2DUP SWAP T>K !   1+ SWAP T>F !   1 TEST_TABLE +!
 LOOP
 DROP ;
1 FILL_TABLE

\ If I=a_record_index & N=I*n_step, each record contains
\ N in the key field & N+1 in the data field.
\ The key fields are ordered small to large, and all data
\ fields hold a non-zero quantity.
```

```
Scr # 9     Forth-83
( speed test                                         06Apr89dna )
: TEST_SPEED  ( #times -- )
 1 ?ENOUGH   BEEP   ." working.."
 0 DO
   256 0 DO
     TEST_TABLE I KEY>FUNC  DROP
   LOOP
 LOOP   BEEP ;
```

BEST OF GENIE

GARY SMITH - LITTLE ROCK, ARKANSAS

News from the GEnie Forth RoundTable—Once again it is time to enjoy some comments from recent GEnie Forth RoundTable guest conferences. Since I am charged with both the privilege of producing this column and arranging the guest conferences, I must admit I truly enjoy these recaps. They give me an opportunity to recall some of the pearls of wisdom I have been audience to, but perhaps failed to properly savor. There are, most definitely, pearls to be gathered.

With the possible exception of conferences such as FORML, Rochester, euroFORML and now the Australian Forth Symposium and SIGForth, I cannot imagine where else one could hope to be exposed to the views of such a variety of Forth luminaries. If you have not participated in one of these conferences, I encourage you to do so. The words remain for your inspection in the GEnie Forth Library, but the intimacy of the moment is missed forever.

For the present moment, sit back and enjoy with me these moments of insight. The guests will be:

- The creators of VP-Planner Plus: Jim Stephens, Kent Brothers, Doug Lankshear, and Chris Worsley.
- Steve Roberts, vagabond computerist and columnist, with John Bumgarner of Information Appliance Inc. and Terry Holmes, the creator of tForth.
- Tom Zimmer, creator of F-PC, the public-domain Forth for PCs with greatly extended features.
- Roedy Green, who created the 32-bit public-domain BBL Forth and Abundance business manager.
- Chuck Moore, Forth's creator and owner of Computer Cowboys.
- Phil Koopman, senior scientist for Harris Semiconductor and author.
- Robert Smith, of Lockheed Palo Alto and Forth math guru.

In the past I have presented the guests' opening remarks, which set the tone of their respective conferences. This format has been well accepted by the readers, so the expression, "If it ain't broke, don't fix it" seems appropriate.

*　　*　　*

Withhold source code only when you're ashamed of it.

Jim Stephenson (with Kent Brothers, Doug Lankshear, and Chris Worsley)
May 1989
Stephenson Software

First, a short blurb about VP-Planner for those who may not know it. VP-Planner is a spreadsheet/database program for the IBM PC, best known for its Lotus 1-2-3 compatible spreadsheet linked with powerful dBASE and multidimensional data-file handling capabilities. It was initially developed in Forth by Jim Stephenson, Dave Mitchell, and Kent Brothers of Vancouver, Canada, and was first released in September 1985 by Paperback Software of Berkeley, California

VP-Planner Plus, released in October 1987, added more database features, 1-2-3 release 2 compatibility, background/priority recalculation, and multi-step undo. The product has been translated into more than ten languages and is sold world-wide. Further development continues on as-yet-unannounced features. The development team now also includes Doug Lankshear, Rick Falck, Bob Tellefson, and Chris Worsley. Jim, Kent, Doug, and Chris have joined the conference this evening to share ideas and answer questions about either VP-Planner or the Forth development system. The Forth system has the following characteristics:

1. direct threaded with `NEXT` coded in-line;
2. top-of-stack in BX register;
3. compiler words and headers in separate area of memory;
4. text in separate area for foreign language translation;
5. colon bodies separated from machine code;
6. hybrid colon/assembly words;
7. local variables and subwords;
8. overlays;
9. extensive Forth-level breakpoint/trace facility;
10. IEEE 64/80-bit software floating point and 80x87 support.

Steve Roberts (with John Bumgarner and Terry Holmes)
June 1989
Freelance writer on tour somewhere on Winnebiko

You probably already know about the Winnebiko, so I won't go into much detail on the general stuff, lifestyle, solar, etc. The emphasis here is on the control system, and I'm delighted to have with me (electronically) John and Terry, who can answer the substantial questions about the new Forth laptop and the details of its implementation. Essentially, I am using this machine as the hub of a real-time control environment in the new bike, in charge of a large "resource bus" that carries all audio, serial, and digital information in the bike. *[The projected release for Information Appliance's Swyft Forth Laptop was first quarter 1990. gls]*

Tom Zimmer
June 1989
Senior Programmer at Maxtor and creator of F-PC Forth

I'm not sure what to say after such a nice intro, but I will say that I am glad to be invited to this round table and for the oportunity to learn more about GEnie. My latest efforts have been in the area of cleaning up F-PC for a new release. The first, and perhaps the most significant, is the adjustment of F-PC to use multiple directories for its sources, rather than keeping five billion files all in one directory. F-TZ, as it is called for the moment, uses a Forth `PATH`, as suggested from the East Coast Forth Board.

Roedy Green
July 1989
Owner of Canadian Mind Products and creator of BBL Forth

There are two sorts of things you probably would be interested in hearing about.

- Internals of the 32 bit BBL Forth compiler.
- Externals of the Abundance database language.

Abundance is more interesting, because I was able to experiment with some novel concepts in languages. BBL is interesting from the point of view of fanatical attention to detail.... Jaunting is the most interesting [feature of Abundance]. It is the ability to run backward in time. Arrays and files use identical syntax. There are no subscripts. Like a spreadsheet, values automatically redisplay on the screen when recomputed.

Chuck Moore
August 1989
Originator of Forth and owner of Computer Cowboys

Pre-conference prelude, the "Future of ..." is a catchall for everything having to do with Forth. Its current place in the world is impossible to determine, and largely irrelevant. Forth is a valuable tool—and will remain so—regardless of the number using it. Recently I was obliged to use conventional CAD software. I am dismayed that it hasn't evolved from the 60's. Forth is the only hope for improved software, ignoring the ever-hopeful AI and neural nets. Computers are getting ever-more complicated, in violation of the first principle of human activity: "Keep It Simple."

In respect for this unique forum—25 words or less—I offer the following statements to challange/guide question/comment:

1. I like classic Forth.
2. This includes `BLOCK`s—simpler, faster, better than files.
3. `VOCABULARY` has been misused by fig-FORTH. It is a poor substitute for fast compile.
4. Forth must evolve. Standards are very dangerous.
5. ANSI committee deserves thanks for "above and beyond call of duty." Theirs is the impossible dream.
6. Marvelous opportunity for non-ANSI Forths.
7. Forth architecture is superb for micro (macro) computers. Many variants should be explored.
8. Three keys are necessary and sufficient. QWERTY is a joke.
9. Marvelous opportunity for non-IBM PCs.
10. Work smart, not hard—forethought.
11. A program that can do everything (ie, `SPICE`) can do nothing well, fast, easily.
12. `PUSH` and `POP` are better names for `>R` and `R>`.
13. Multiply is a much-over-used arithmetic operation (i.e., FFT can be replaced by Walsh-Hadamard).
14. Floating point is a bad joke.
15. Withhold source code only when you're ashamed of it.

Forth is the best computer language. I'll be using it another 20 years, with a few changes.

Phil Koopman
September 1989
Senior Scientist, Harris Semiconductor and author of Stack Machines: The New Wave

Some of the things I have found out about stack machines go against widely held (at least, outside the Forth community) ideas. For example, stack machines:

- don't need stacks bigger than 16 to 32 elements
- need not have a significant context-switching time
- can cycle their clocks every bit as fast as (or perhaps faster than) RISC processors

One thing I run across continually is that folks confuse the requirements for real-time embedded control with those of workstation environments. One of my professional goals is to understand more about Forth-derived stack computers in order to help them gain acceptance in applications for which they are well suited. Stack machines seem to be superb at real-time embedded control (although I still want to do more research to quantify this notion). But, what about other application areas? If stack machines are the answer, what are the questions?

Robert L. Smith
October 1989 Research
Specialist with Lockheed, Palo Alto

Thank you. For floored division, it helps to focus on the modulus or remainder rather than the quotient. Most users use only positive arguments, so floored or non-floored give the same results. For almost all cases that I know of, if you have at least a negative numerator, you probably should use floored division.

As for floating point:

1. Should Forth have it at all?
2. If so, should it be in the Standard?
3. [Should it be] IEEE floating point?

*   *   *

It is never too late to begin participation in the guest conferences. They are usually scheduled for the third Thursday of the month except for the last three months of the calendar year, when they are scheduled for the second Thursday to avoid conflict with the holidays. Obviously there are exceptions, so it is always wise to note the current schedule that appears each day you log onto the GEnie Forth RoundTable. I might add that without attendees (with questions) it is pointless to schedule these wonderful guests.

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

(Continued from page 15.)

next command.

Screens 1–5 represent the first steps of this development, 6–9 are the second stage, and 10–11 comprise the final stage.

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.

(Page screens, continued.)

```
TRACE SCR # 8
\ Second stage, continued                           30JUL88CHP
ASSEMBLE (DETOUR)
   SEC, IP LDA, 2 # SBC, FROM STA, IP 1+ LDA, 0 # SBC,
   FFFF CMP, 101 BCC, 102 BNE, FROM LDA, FFFF CMP,
   101 BCC,
   102 DEX, DEX, W LDA, PHA, 0 ,X STA,
              W 1+ LDA, PHA, 1 ,X STA,
\ Directions for detour
   FF # LDA, W 1+ STA,
   FF # LDA, W STA,
\ Remove detour signpost
   F0 # LDA, ' NEXT 19 + STA, 0 # LDA, ' NEXT 1A + STA,
   101 0 # LDY, W 1- JMP, END   -->
END sets the branches to the labels 101, 102, etc.
103 is a dummy label; /MOD puts two numbers on the stack,
the first would be misinterpreted as a label if no label

TRACE SCR # 9
\ Second stage, concluded                           30JUL88CHP

: TRACE ['] (DETOUR) 2+ ['] NEXT 19 + ! ;
: NOTRACE F0 ['] NEXT 19 + ! ;

 ' (DETOUR) 2+ 100 /MOD ' (UNDETOUR) F + C!
                        ' (UNDETOUR) 14 + C!

FLOOR 1+ ' (DETOUR) E + !
FLOOR ' (DETOUR) 17 + !
' DETOUR 100 /MOD ' (DETOUR) 28 + C! ' (DETOUR) 2C + C!

' DUMMY >NAME ' FLOOR >LINK !
\ Establishes a link bypassing the assembler
HERE FLOOR !
QUIT

TRACE SCR # 10
\ Third stage                                       30JUL88CHP
HEX
VARIABLE FLOOR
: INDENT 24 C! ; \ APPLE specific
: DISPLAY 0 HEX <# # # # #S #> TYPE 2 SPACES DECIMAL ;
: .S C INDENT DEPTH ?DUP IF 0 DO DEPTH I - 1- PICK DISPLAY
    LOOP ELSE ." Empty stack" THEN ;
:  PRIM -2 ALLOT HERE 2+ , ;
\ PRIM converts the execution procedure (installed by CREATE)
     \ from that of a variable to that of a primitive
CREATE (UNDETOUR) PRIM 8568 , 68EE , EF85 , 8568 ,
    68F2 , F185 , FFA9 , 248D , A909 , 8DFF , 0923 ,
    00A0 , F04C , 00 C,
: NOTRACE F0 ['] NEXT 19 + ! ;
-->
NOTRACE moved up to allow the new DETOUR on next screen

TRACE SCR # 11
\ Third stage, concluded                            30JUL88CHP
CREATE (DETOUR) PRIM A538 , E9EE , 8502 , A5F4 ,
  E9EF , CD00 , FFFF , 2790 , 07D0 , F4A5 ,
  FFCD , 90FF , CA1E , A5CA , 48F1 , 0095 ,
  F2A5 , 9548 , A901 , 85FF , A9F2 , 85FF ,
  A9F1 , 8DF0 , 0923 , 00A9 , 248D , A009 ,
  4C00 , 00F0 ,
: DETOUR >R .S KEY DUP 7F = IF ABORT THEN 20 = 0=
     IF R> DROP R> R> DROP 2- >R NOTRACE CR EXIT THEN
     R> CR >NAME ID. 4 SPACES (UNDETOUR) ;
: TRACE ['] (DETOUR) 2+ ['] NEXT 19 + ! ;
' (DETOUR) 2+ 100 /MOD ' (UNDETOUR) F + C!
                       ' (UNDETOUR) 14 + C!
FLOOR 1+ ' (DETOUR) 0E + !      FLOOR ' (DETOUR) 17 + !
' DETOUR 100 /MOD ' (DETOUR) 28 + C! ' (DETOUR) 2C + C!
HERE FLOOR !  QUIT
```

SEEING FORTH

JACK J. WOEHR - 'JAX' ON GEnie

"Remontons vers les faits moins visibles, mais plus importantes. Nous y verrons le retour á l'âge des Adeptes."

—Louis Pauwels and Jacques Bergier
Le Matin des Magiciens
Editions Gaillimard, 1960

The Grand Adept of Forth was and remains Charles Moore himself, whom some describe as the author of Forth and others as the discoverer of same.

Charles Moore is a tall, smiling, pleasant man in his forties with neat, dark hair and a balding dome which he covers with a tasteful cowboy hat. He also wears cowboy boots and is associated with a firm called Computer Cowboys.

Forth idealizes an imaginary processing unit.

Mr. Moore characterizes himself as "the one you can blame for all this." In a sense he is correct; a wind of freedom blows from the direction of Forth that is most disconcerting to those trapped in jobs which mandate the use of a traditional compiler.

Moore is cryptic when asked to describe his invention. He is a habitual iconoclast, as delighted at bursting the bubbles of his disciples as of his opponents.

"Forth, to me, is more of an approach than a specification for a programming language," he says when asked his opinion of attempts to standardize Forth.

Let us examine that approach.

Forth idealizes an imaginary processing

```
\ scasm32.f ...
\ assembler for SC32 in JForth
\ ©1989 jack j. woehr
\ permission to distribute and use freely granted
\ to Forth Interest Group MEMBERS ONLY !!!
\ pay yer dues, cheapskate!
\ and attend your local FIG Chapter regularly!
\ jax@well.UUCP JAX on GEnie
\ Minimal instruction assembler written in JForth for the Johns Hopkins
\ JPL 32-bit stack machine known as the SC32.
\ references:
\         Silicon Composers, Inc., 32-Bit
\         Stack-Chip Microprocessor Preliminary,
\         4/12/89.
\         ©1989, Silicon Composers, Palo Alto, CA
\ Examples:
\
\      CALL 1234567 ADDRESS ,
\
\ ALU/SHIFT NEXT U3 SOURCE S0 DEST PUSHS-POPR STACK 0 CIN DST<ALU BUSSRC
\      V ALUCOND FL<ALUCOND FLAG S0&SRC ALU ,
\

hex

only forth definitions also

vocabulary SCASM32

also SCASM32 definitions also

\ *** Instruction Logic

\ Instruction Types

00000000 constant call
20000000 constant branch
40000000 constant branch?
60000000 constant ALU/shift
80000000 constant load
A0000000 constant store
C0000000 constant load-addr-low
E0000000 constant load-addr-high
```

```
\ convenient op name aliases

load-addr-low  constant lal
load-addr-high constant lah

\ Next Bit

10000000 constant  #next

\ Src & Dst Bit Patterns

14 constant dst-field
18 constant src-field

00 constant s0
01 constant s1
02 constant s2
03 constant s3
04 constant r0
05 constant r1
06 constant r2
07 constant r3
08 constant u0
09 constant u1
0A constant u2
0B constant u3
0C constant pc
0D constant psw
0E constant zero

 ( 0F reserved)

 \ Stack Bit Patterns

 10 constant stack-field

 00 constant nop \ This also applies for the Flag Field of the ALU/
 Logic ops.
 01 constant popr
 02 constant pushr

 ( 03 reserved)

 04 constant pops
 05 constant pops-popr
 06 constant pops-pushr

 ( 07 reserved )

 08 constant pushs
 09 constant pushs-popr
 0A constant pushs-pushr

 ( 0B - 0F reserved )

 \ ALU/Shift

 \ Fields

 0F constant subtype-field
 0E constant bussrc-field
 0A constant alucond-field
 08 constant cin-field
 07 constant flag-field
 00 constant alu-operation-field
```

(Continued on next page.)

unit with an infinitely extensible instruction set. Such a processor not yet existing, Forth is asymptotic to the progress of Forth implementations. So we see that where Moore appears frustratingly vague to his eager hearers, he is actually being explicit.

If Moore is an adept, he must have a lineage. Dr. C.H. Ting, himself a Forth adept, compares the CISC (Complicated Instruction Set) style of Forth with the available academic models and proclaims Moore heir to Von Neumann. Von Neumann and his associates gave contours to serial computation conducted by electronic digital devices which held near-universal sway until recent years. Now the Harvard architecture rears up in belated challenge as we sit on the threshold of the parallel-computation age; but it is significant that the retooling of Von Neumannism inherent in Forth is of an age equal to the Harvard model, and it has progressed to a greater variety of implementations ahead of the evolution of the Harvard model, the latter requiring a much greater silicon investment before its benefits could be made manifest.

Forth, from its inception, has been remarkably easy to implement on a certain level. This was one of its most attractive points to early enthusiasts who found themselves in a race with rapidly changing hardware in the computer explosion of the seventies and early eighties. Forth seems alive; once "life" has been established—once a nucleus of indispensable instructions has been coded—the system awakens and begins to grow beneath the sculpting hand of the programmer.

The real-world emulations of the ideal Forth have culminated in our time with microprocessors specifically designed to execute the fundamental Forth instruction set. Yet Forth itself remains elusive, almost reticent, much like Moore himself. Perhaps we have come as close to the Muse as she will allow us to approach in this Digital Dispensation, and we shall now be forced to take refuge in standards, and in technique.

(Multitasking, continued from page 18.)

```
20 CONSTANT SCORE
```

The 20 (like all numbers) is placed on the stack, then CONSTANT is activated. CONSTANT is a defining word, and a defining word is always followed by a name to give to the 'thing' it is to define (in this case SCORE). CONSTANT places this name in the dictionary, reserves space for one number, and installs the number on the stack in this space. This completes the building; it then adds instructions about the run-time behavior of SCORE.[1] All constants have the same run-time behavior, which is to place on top of the stack the number stored as part of their structure.

CONSTANT could have been defined using the words CREATE (which starts the instructions on what to build) and DOES> (which starts the list of run-time behavior instructions) as follows:[2]

```
: CONSTANT
  CREATE , DOES> @  ;
```

CREATE starts the building process by adding a name to the dictionary, using the next word in the input string (the word after CONSTANT) for the name. The , (comma) reserves two bytes and initializes them by storing the number from the top of the stack at the end of the dictionary and advancing the dictionary pointer (the pointer to the next available free space at the end of the dictionary). DOES> starts the series of run-time behavior instructions with the minimum action, which is to return the address of the first thing CREATE built after the name. In the case of a constant, this is the address of the stored value, so the only other action needed is to read the value stored there with a normal fetch.

Returning to our new defining word, CREATE and DOES> are used to define the two functional parts of TIMER, as shown in Figure One. TIMER builds a name and the space for two 16-bit variables, the user value UV and the internal value IV. The run-time behavior given to the word defined by TIMER is to put the address of the user variable on the stack and read the real-time clock. Then the last value read is subtracted and the user variable is corrected.

[1]To be picky, F83 does not place the instructions there, it places a pointer to instructions. However, this is a point of implementation detail that can be ignored here.

[2]It isn't in most systems—it is defined as a primitive in the interests of speed—but it could have been.

(Continued.)

```
\ SubT Field Bit Patterns

0 constant alu/logic
1 constant shift/step

\ BusSrc Field Bit Patterns

0 constant dst<fl
1 constant dst<alu

\ ALU Condition Field Bit Patterns

( 00 constant 0) \ These conveniently are unambiguously themselves!
( 01 constant 1) \ Likewise with the Cin instructions.

02 constant V
03 constant _V
04 constant _((NxV)|Z)
05 constant (NxV)|Z
06 constant N
07 constant _N
08 constant Z
09 constant _Z
0A constant _(_C|Z)
0B constant _C|Z
0C constant NxV
0D constant _(NxV)
0E constant C  \ watch out with the hex numbers, always precede w/ 0 !!
0F constant _C

\ Cin Field Bit Patterns

( 00 constant 0) \ Conveniently, unambiguously themselves ...
( 01 constant 1) \ ... as w/ ALU Conditions above

02 constant FL'
03 constant _FL'

\ Flag Field Bit Patterns

( 0 constant nop ) \ Same as above in the Stack Field Bit Patterns

1 constant fl<alucond

\ ALU Operations

15 constant _(s0&src)
17 constant s0|_src
1D constant _s0|src
1F constant s0|src
20 constant 0
21 constant _s0
22 constant negl
23 constant s0
24 constant _src
2C constant src
2F constant s0xsrc
41 constant _s0+cin
43 constant s0+cin
44 constant _src+cin
45 constant _s0+_src+cin
46 constant _src-_cin
47 constant s0-src-_cin
49 constant _s0-_cin
```

```
4B constant s0-_cin
4C constant src+cin
4D constant src-s0-_cin
4E constant src-_cin
4F constant s0+src+cin
55 constant s0&src
57 constant _s0&src
5D constant S0&_src
5F constant _(s0|src)
6F constant _(s0xsrc)

\ Shift Instructions
\ Shift Fields

5 constant shift-field
4 constant shiftin-field
2 constant step-field
1 constant flagin-field

\ Shift Field Bit Patterns

\ Shift

( 0 constant nop)\ Once again, this is conveniently already defined

1 constant right
2 constant left

\ Shiftin

0 constant <alucond
1 constant <FL'

\ Step

0 constant step:src+cin
1 constant step:src-s0-cin
2 constant step:s0+src+cin(FL')
3 constant step:s0+src+cin(_FL')

\ Flagin

( 0 constant <alucond)        \ already defined above in Shiftin

1 constant <shiftoutput

\ *** Forming Instructions

\ Control Flow

: address     \ control-instruction address -- instruction'
      1FFFFFFF and or ;

\ Shifting Bit Pattern to Instruction Field

: shift-into-field    \ instruction bits field -- instruction'
      << or ;

\ Set Next Bit

: next \ instruction -- instruction'
      #next or ;

\ Src & Dst
```

(Continued on next page.)

The last value read is then updated, and we exit with the address of the user variable still on the stack.

A definition for `(READ_CLOCK)` to suit the IBM PC and F83 is given in Figure Two; it returns a number which is incremented 1193180/65536 times per second (a strange number, granted, but that is how IBM designed it). After this (or a substitute that suits your hardware) and `TIMER` are entered, the following can be used as a test:

```
TIMER CLOCK
: TEST
  BEGIN CLOCK
  @ DUP U. 0<= UNTIL
  ." Timed out!"  ;
```

Then, if you enter the line:

```
180 CLOCK ! TEST
```

a series of decreasing numbers (the user variable) will be printed—which lasts just under ten seconds on my system—before the "Timed out!" message appears.

To complete the task, the multitasker must be used. Multitasking has been part of almost all versions of Forth except fig-FORTH, the first of the public-domain versions. It is not, however, part of the standard. Unlike time-sliced multitasking, in which each task has to surrender the processor to the next task after a pre-determined time interval whether it "likes" it or not, F83 (like most versions of Forth) uses a cooperative scheme. In this, a task passes control only when it is ready, thus simplifying the job of keeping track of who is doing what, and making the task interchange very fast. The cost is that one cannot predict reliably exactly when task interchange will take place, and if one task gets into an endless loop that does not contain the voluntary transfer word `PAUSE`, everything else stops for good. This latter case is the fault of the programmer, not the language. With care, the task latency time can be made very small, especially since all F83 words having to do with human interaction—and whose execution times are, therefore, unpredictable—already contain the task interchange word `PAUSE`.

Different tasks share all resources other than the stacks, although a group of variables has to be assigned to each task to keep a record of internal processor information during the time when other tasks have control. The tasks involved in the multitasking are linked into a circular list, each receiving control from the preceding one and passing it to the succeeding one. Each task on the list can be active or asleep. In the latter state, it passes control on as soon as it receives it. Otherwise, it executes until the word `PAUSE` is encountered, either explicitly or as part of an input or output word. A task can be activated by use of the word `AWAKE`. Multitasking can be turned off or on by the words `MULTI` and `SINGLE`. If absolutely essential, these could be used within a task if, for some reason, the task had to retain control for a certain period even though some input or output words (which would normally cause a task interchange) are to be executed.

The user-interface words involved in multitasking in F83 are given in Figure Three. The use of these words is demonstrated in Figure Four. First, we use the special defining word `TASK:` to build a task that prints 20 asterisks on the screen and link it into the round robin (which at the moment only consists of the outer interpreter, which is handling our keyboard input).

Note that the `STOP` is essential. Otherwise, when 20 asterisks have been printed and the task is over, disaster will strike as the computer tries to execute the stack for `PRINT*S`! Also note that we have an inner loop just to slow things down a bit, otherwise all the asterisks will appear before we have a chance to do anything. This inner loop is a good neighbor and gives everyone else a go by, including the word `PAUSE` in the loop.

Nothing unusual happens on the screen, as we have not turned on multitasking. We can change that easily by entering `MULTI`. Still no asterisk appears; this is because when a task is built and linked, it is placed in the sleeping condition. Hence, we must enter `PRINT*S WAKE` to wake it up. We can carry on typing at the keyboard, but on the screen our input will appear mixed with asterisks. Well, it will until 20 asterisks are printed, then things will return to normal.

Entering `PRINT*S  WAKE` again will *not* cause another batch of asterisks to appear. The task will resume with the (nonexistent) word after `STOP`, and disaster will strike. As it stands, `PRINT*S` is a one-shot model only!

If, during this batch of asterisks, we had managed to type

```
PRINT*S SLEEP <cr>
```

the output of asterisks would have stopped at once. The same would happen if we were

(Continued.)

```
: source      \ instruction register -- instruction'
      src-field shift-into-field ;

: dest \ instruction register -- instruction'
      dst-field shift-into-field ;

\ Stack Action

: stack\ instruction stackop -- instruction'
      stack-field shift-into-field ;

\ Load and Store

: zero-extended-offset  \ instruction addr-offset -- instruction'
      0ffff and or ;

: zeo   \ i a-o -- i'  \ convenient alias
      zero-extended-offset ;

\ ALU/Shift Instructions

\ ALU Operations

: subtype     \ instruction subtype -- instruction'
      subtype-field shift-into-field ;

: bussrc      \ instruction bussrc -- instruction'
      bussrc-field shift-into-field ;

: alucond     \ instruction type -- instruction'
      alucond-field shift-into-field ;

: cin   \ instruction type -- instruction'
      cin-field shift-into-field ;

: flag \ instruction type -- instruction'
      flag-field shift-into-field ;

: alu   \ instruction operation --
      alu-operation-field shift-into-field ;

\ Shift Operations

: shift\ instruction shift-op -- instruction'
      shift-field shift-into-field ;

: shiftin     \ instruction shifter-input -- instruction'
      shiftin-field shift-into-field ;

: step \ instruction step-op -- instruction'
      step-field shift-into-field ;

: flagin      \ instructions flagin --
      flagin-field shift-into-field ;

\ *** Assembly Buffer Management

   4 constant buff-hdr-size  \ link to next allocated buffer | 0
1000 constant buff-size      \ each buffer same size
   4 constant opsize         \ each op is a longword on SC32

variable 1st-buffer          \ as it says
variable last-buffer         \ latest allocated buffer
```

```
variable asm-ptr   \ the "dictionary pointer" for our assembler

: asm-ptr.init \ --     \ start off set to zero
      asm-ptr off ;

: here          \ -- next-available-assembly-relative-address
      asm-ptr @ ;

: (there)              \ addr -- offset-in-any-buffer
      buff-size mod ;

: there          \ here -- real-address
      (there) last-buffer @ buff-hdr-size + + ;

\ allocate $1004 bytes for assembly and a buffer-linking header.

: buff-err?    \ --
      abort" No Buffer Memory" ;

: get-buffer   \ -- absmemaddr|0
      [ buff-hdr-size buff-size + ] literal MEMF_CLEAR
      [ forth ] exec? call exec_lib allocmem [ scasm32 ] ;

: free-buffer  \ reladdr --
      >abs
      [ buff-hdr-size buff-size + ] literal
      [ forth ] call exec_lib freemem drop [ scasm32 ] ;

: free-all-buffers    \ --
      1st-buffer @
      begin
      dup @ swap free-buffer
      dup 0=
      until drop ;

: get-1st-buffer      \ --
      get-buffer dup
      if >rel dup 1st-buffer ! last-buffer !
      else true buff-err?
      then ;

: get-subsequent-buffer       \ --
      get-buffer dup
      if >rel dup last-buffer @ ! last-buffer !
      else true buff-err?
      then ;

: manage-buffers      \ --
      here (there) 0= here 0> and
      if get-subsequent-buffer then ;

\ *** Output File Handling

create out-filename 100 allot
variable out-fileptr
variable outfile-buff

: outfile-err? \ t|f --
      abort" Couldn't Open Output File" ;

: writefile-err?       \ t|f --
      abort" Error While Writing Output File" ;

: out-filename.default       \ --
      0" ram:scasm32.out" 0count 1+ out-filename swap cmove ;
```

(Continued.)

to type SINGLE, although this would "lock" the processor on the one task in which it occurred. The other tasks would not be asleep, but would start as soon as MULTI was issued, without having to be awakened. If the task in which the SINGLE command occurred didn't have MULTI later, or had no way of inputting the MULTI command (i.e., didn't involve the outer interpreter), there would be no way short of a system reset to regain control.

BACKGROUND: adds a new task into the round robin. How can one remove one that is no longer needed? The simple answer is, you cannot. You can assign new instructions to the old task name, but you must not FORGET the old task, as the circular list would be broken and disaster would strike when the processor tried to move around it. To assign a new set of instructions, the word ACTIVATE is used to associate the new instructions with the old name and wake it up immediately. We will use ACTIVATE to assign a new version to PRINT*S, one which will be reusable. It is essential to realize that ACTIVATE can only be used in a colon definition, because of the way it handles the return stack; attempts to use it interactively will cause a system crash.

The version of our example shown in Figure Five is much better; when awakened after running to completion, it just loops and runs again. The original version is replaced by this improvement just by executing NEW-PRINT*S.

For speed, you should have only enough tasks in the circular list to service the maximum number that must run concurrently, and use task redefinition to move tasks into and out of the list. Task interchange is fast, but it doesn't take zero time.

Vast possibilities arise from the ability to run tasks, freeze them, and later restart them; for tasks to start and stop other tasks; and for tasks to be able to grab all the processing power for time-critical routines by issuing SINGLE and, afterwards, MULTI. However, the virtues of simplicity are nowhere stronger than in multitasking. All tasks must cooperate, and the problem of keeping in mind the possible effects of all combinations of events rapidly becomes daunting.

I do not like the name used for the word BACKGROUND:, as it suggests to me a master-slave relationship rather than a cooperative arrangement. Also, the allocation of 400 bytes is not always ideal, although 100 bytes of whatever quantity you allocate will go for the return stack and the rest for the data stack (unless you change `TASK:`). Of course, in the spirit of Forth, if you don't like it, change it.

The formal definition of `BACKGROUND:` is given in Figure Six. It is a short definition, and it is easy to modify the number of bytes required for the two stacks. After modification, it could be saved as `MULTITASK` or `COOP-MEMBER` or any other name which takes your fancy. Similarly, I would prefer `IS-NOW` for `ACTIVATE`, but that is a personal matter. If you wish to change the name, it can easily be done with:

```
: IS-NOW  ACTIVATE ;
```

which make the two names mean the same. If you wish to allocate more or less than 100 bytes to the return stack, you will need to redefine `TASK:` and then use your new definition in a new version of `BACKGROUND:`. To find where to change `TASK:`, decompile (e.g., `SEE TASK:`) and then re-enter it, changing the 100 just past halfway through the definition to whatever number you prefer. The data stack will get the difference between what you put in your version of `TASK:` and the total allocation for stacks you define in your version of `BACKGROUND:`.

Interrupts will be needed for very rapidly occurring events but, for most other situations, the timer and multitasker described above will give you real-time control. For further detail on the multitasker in F83, see the shadow screens of the source code or chapter 23 of *Inside F83* by C.H. Ting.

Tim Hendtlass is principal lecturer in scientific instrumentation in the physics department of the Swinburne Institute of Technology. He discovered Forth in 1980, used it in more and more instrumentation, and introduced it as the laboratory language for all undergraduate students majoring in scientific instrumentation.

(Continued.)

```
: open-outfile \ --    \ what the heck is going on here?
      out-filename new 0fopen
      dup out-fileptr !
      0= outfile-err?
      out-fileptr @ fclose
      out-filename 0fopen dup
      0= outfile-err?
      out-fileptr ! ;

: close-outfile       \ --
      out-fileptr @ fclose ;

: (get-outfile-buff)  \ -- 0|abs_addr
      here MEMF_CLEAR
      [ forth ] exec? call exec_lib allocmem [ scasm32 ] ;

: get-outfile-buff    \ --
      (get-outfile-buff) dup 0= abort" Couldn't Get Outfile Buff"
      >rel outfile-buff ! ;

: free-outfile-buff              \ --
      outfile-buff @ >abs here
      [ forth ] call exec_lib freemem drop [ scasm32 ] ;

: fill-outfile-buff              \ --
      here buff-size /mod        \ how many 4k buffs to consoli-
date?
      outfile-buff @             \ get the file buffer
      1st-buffer @ rot           \ get the first asm buffer
      0 do                       \ JForth DO is a ?DO
              dup buff-hdr-size + \ move to data area
              2 pick             \ get file buffer address
              buff-size cmove>   \ move data
              @ swap buff-size + swap      \ get next data buff,
inc file buff adr
          loop
      rot dup                    \ is there a modulus remaining?
      if                         \ yes, copy rest of data
              swap buff-hdr-size + -rot cmove>
      else
              2drop drop         \ drop data-adr filebuf-adr 0ct
      then ;

: write-outfile       \ --
      out-fileptr @ outfile-buff @ here fwrite
      0= abort" Error Writing Assembly to File" ;

: save-assembly       \ --
      open-outfile
      get-outfile-buff fill-outfile-buff
      write-outfile close-outfile free-outfile-buff ;

\ *** Assemble to Memory

\ all SC32 operands are longwords

: ,          \ longword --
      manage-buffers here there ! 4 asm-ptr +! ;

\ *** Initialization

: scasm32.init \ --
      scasm32 get-1st-buffer asm-ptr.init out-filename.default ;

: wrapup              \ --
      save-assembly free-all-buffers ;

decimal
```

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

(Continued on page 37.)

FIG CHAPTERS REPORT

JACK WOEHR - 'JAX' ON GEnie

The British Columbia Forth Interest Group Chapter has been having a very lively year. Their high-power sessions have included an address by Soviet Forther Serge Baranoff. Here are the minutes of a recent BC-FIG meeting.

* * *

Minutes of the BC-FIG Chapter
October 5, 1989, 7:30 p.m.
Place: BCIT, Burnaby, B.C., Canada
Attended by: John Somerville, Gordon Ganderton, Nick Janow, Doug Lankshear, Zafar Essak, Kenneth O'Heskin, Paul Unruh, Jack Brown, and Dave Brown

Robot

The first item on the agenda was an update by Jack Brown on the progress of the robot-building course which four members of the chapter are taking. Jack displayed the hardware, and reported that the course is well designed and organised, with good support from their instructor. For example, when the students assembled their boards they were able to test them on the instructor's working robot, and any problems could be diagnosed and fixed on the spot. Jack also mentioned when he has his machine up and running (or down and whirring and clicking—the device will end up looking like a mobile teakettle, probably rather menacing to a cat), he'll retrofit it with a Forth engine. It's obvious the participants in the course are having a good time.

Pocket Forth Computer

John Somerville demonstrated a vintage Hewlett-Packard machine which contains many interesting features, not the least of which is Forth. Although no longer supported (one is reminded that obsolete technology, or what never did make it in the marketplace, is often inherently interesting; cf. recent exchanges on the Forth nets about the Jupiter Ace), the machine has room for add-on 64K memory modules, I/O ports, 20-bit addressing on a proprietary H-P CPU—in other words, an early laptop in a hand-calculator box. The kicker of John's demonstration was that, although it boots up in BASIC, Forth can be called as a "subroutine" and, when in Forth mode, BASIC can be called as a subroutine of Forth!

Fifth 2.7

Jack Brown put Fifth (shareware version) through its paces, which revealed itself as not too unfamiliar to Forth users, although different enough to require the manuals and tutorials. Jack pointed out that some impressive application software has been written in Fifth, and some attendees expressed interest in checking it out further.

The meeting adjourned for coffee and conversation.

* * *

About two years back, the Silicon Valley FIG Chapter, which was meeting at Hewlett-Packard, decided that bay area interest in FIG activities was great enough to split up into North Bay and South Bay FIG Chapters. A move from the traditional H-P site and declining attendance in the North Bay are forcing the leaders to take a second look at their historic decision. As of this reading, the die may already be cast for the re-merger of two of the most exciting FIG groups in the world. If you are interested in the preservation of both chapters, "vote with your feet" and help increase attendance in the Bay area.

A new nationwide FIG Chapter is in the works for Spain. The interested parties recently contacted the Forth Interest Group and informed us that they could justify the existence of a FIG Chapter if it could be considered a national group, rather than regional, to which we gave our happy assent. Interested parties should contact:

Borja Marcos
Alangoeta, 11, 1ro izq.
48990 - Algorta (Vizcaya)
Spain

We are informed that a persistent problem with FIG Chapters continues unabated: that is, the moving and/or disappearance of Chapters without forwarding addresses or notifications of the Forth Interest Group. This problem, and the perception on our part that the central organization is losing (has lost?) contact with the needs of the membership, prompts us to undertake a simple experiment.

After this issue of *Forth Dimensions* is published, the Chapter Coordinator (presumably still the author by that time) will telephone around to all the North American chapters and try to verify their existence and get an introduction to their coordinators and insight into their operation.

Please notify me, if possible, if there is a time when you (i.e., the contact party listed in the directory at the back of this magazine) would prefer to be contacted. I can be reached during working hours at 303-422-8088. My computer bulletin board is 303-278-0364 (300/1200/2400, 24 hours). My email addresses are jax@well{.UUCP, .sf.ca.us} and FIGCHAPTERS or JAX on GEnie. My mailing address is:

Jack Woehr
Vesta Technology, Inc.
Suite 101
7100 W. 44th Ave.
Wheat Ridge, Colorado 80033

I look forward to chatting with as many of you as I can reach, as we work together to set the agenda for the Forth Interest Group for the new decade.

(Reference Section continued)

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)

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

International Forth BBS's

- Melbourne FIG Chapter
  (03) 299-1787 in Australia
  61-3-299-1787 international
  SysOp: Lance Collins
- Forth BBS JEDI
  Paris, France
  33 36 43 15 15
  7 data bits, 1 stop, even parity
- Max BBS *(ForthNet link*)*
  United Kingdom
  0905 754157
  SysOp: Jon Brooks
- Sky Port *(ForthNet link*)*
  United Kingdom
  44-1-294-1006
  SysOp: Andy Brimson
- SweFIG
  Per Alm Sweden
  46-8-71-35751

This list was accurate as of October 1989. If you know another on-line Forth resource, please let me know so it can be included in this list. I can be reached in the following ways:

Gary Smith
P. O. Drawer 7680
Little Rock, Arkansas 72217
Telephone: 501-227-7817

GEnie (co-SysOp, Forth RT and Unix RT): GARY-S
Usenet domain:
uunet!wugate!
wuarchive!texbell!
ark!lrark!gars

**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.*

Advertisers Index

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
  Tom Ghormley
  (916) 444-7775

  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
  4th Thurs.
  Tom Chrapkiewicz
  (313) 322-7862

- **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.)

1990 ROCHESTER FORTH CONFERENCE ON EMBEDDED SYSTEMS

June, 1990
University of Rochester
Rochester, New York

Call for Papers

There is a call for papers on the use of Forth technology in Embedded Systems. Papers are limited to 5 pages, and abstracts to 100 words. Longer papers will be considered for review in the refereed Journal of Forth Application and Research.

Please send abstracts by March 15, 1990 and final papers by May 15, 1990.

For more information, contact:

Lawrence P. Forsley
Conference Chairman
Institute for Applied Forth Research, Inc.
70 Elmwood Avenue
Rochester, NY 14611
(716)-235-0168 • (716)-328-6426 (FAX)

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

Second Class
Postage Paid at
San Jose, CA