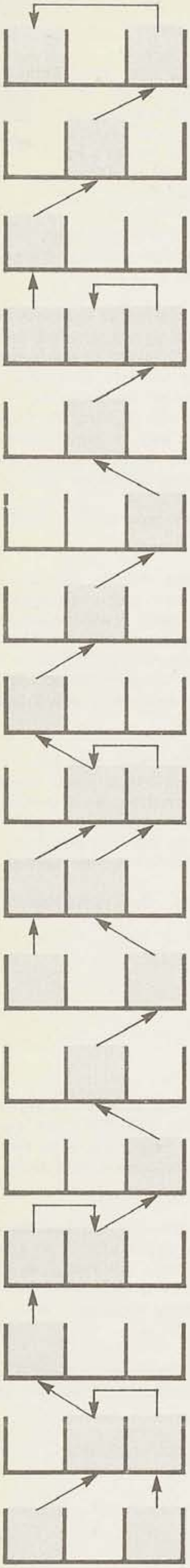


FORTH DIMENSIONS

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

Volume III
Number 6
Price: $2.50

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume III No. 6 March/April 1982

Publisher Roy C. Martens
Editor Leo Brodie

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. MOST MATERIAL PUBLISHED BY THE FORTH INTEREST GROUP IS IN THE PUBLIC DOMAIN. Information in FORTH DIMENSIONS may be reproduced with credit given to the author and the Forth Interest Group.

Subscription to FORTH DIMENSIONS is free with membership in the Forth Interest Group at $15.00 per year ($27.00 foreign air). For membership, change of address and/or to submit material, the address is:

Forth Interest Group
P.O. Box 1105
San Carlos, CA 94070

HISTORICAL PERSPECTIVE

FORTH was created by Mr. Charles H. Moore in 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of dissatisfaction with available programming tools, especially for observatory automation.

Mr. Moore and several associates formed FORTH, Inc. in 1973 fo the purpose of licensing and support of the FORTH Operating System and Programming Language, and to supply application programming to meet customers' unique requirements.

The Forth Interest Group is centered in Northern California. Our membership is over 3,500 worldwide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

PUBLIC NOTICE

Although the FORTH Interest Group specifies all its publications are non-copyright (public domain), several exceptions exist. As a matter of record, we would like to note that the copyright has been retained on the 6809 Assembly listing by Talbott Microsystems and the Alpha-Micro Assembly listing by Robert Berkey. Several conference papers have had copyright reserved. The general statement by FIG cannot be taken an absolute, where the author states otherwise.

FROM THE EDITOR

Hi! I'm happy to say that starting with this issue, I'll be serving as regular editor of FORTH Dimensions. I'd like to thank Carl Street, the previous editor, who has been a great help to me during the transition. Carl has made several important contributions to FORTH Dimensions, such as the writer's kit for helping you submit articles. Carl will rejoin FORTH Dimensions as our advertising director beginning later this year.

I'd also like to thank Roy Martens, the publisher, for suggesting that I take the editor's post, and for teaching me some of the facts of life in magazine publication.

I hope to make this magazine as useful as possible to the greatest number of people. Since most of our readers are still learning FORTH at one level or another, I intend to encourage the publication of tutorials (such as Henry Laxen's excellent series which continues with this issue), application stories (sure, FORTH is fun, but let's show the world what we can do with it!), examples of well-written FORTH code (the best way to learn style is by reading elegant examples), and any ideas, discoveries, impressions or feelings you care to express (this is your magazine, after all!).

In short, we'll be concentrating on how to use FORTH in solving problems.

By contrast, system implementation details are more the responsibility of the individual vendors' documentation. In addition, the FORTH community boasts two organizations devoted to improving and extending the language: the Standards Team and the FORTH Modification Laboratory (FORML). Each of these groups convenes annually, and the proceedings of these conventions (available through FIG) are extremely valuable documents for the advanced study of FORTH.

I'm looking to each of you to help make this the kind of magazine you want it to be, by contributing articles, examples, and letters. We don't have a staff of writers, so everything we print comes from you. (If you want to contribute but don't know what or how, drop me a line. I'll send you the information kit that Carl put together, and answer any questions you may have.)

I hope you enjoy FORTH Dimensions. And remember, I hope to hear from all of you.

Leo Brodie

NEW POLICY

The 79-Standard has been voted on and adopted to serve as a common denominator for transportable FORTH code and for future discussion of FORTH systems. Beginning with the next issue, FORTH DIMENSIONS will give preference to articles that adopt the 79-Standard

Listings which use words that are not 79-Standard are welcome, but if possible explain such words in a brief glossary with a note that they are not 79-Standard. For instance, if your application addresses the name field of a definition (which is illegal in the Standard), you should supply a glossary description of NFA.

If possible, also include the definition of such a word. High level source is preferred, but if necessary, the definition may be written in assembler.

We hope this policy will encourage unification, eliminate ambiguity, and simplify explanations.

LETTERS

FORTH Application Library

Dear fig,

As distributors in the UK for FORTH Inc., with a rapidly growing customer base, we are potentially interested in any application software that is generally useful.

Most of our customers are in the process control/industrial/scientific sectors which, by their nature, require fairly specialized and customized software. Nevertheless, we are sure there are many areas of commonly useful software and that such software would be useful even if only as a starting point or guideline, in order to avoid too much reinvention of the wheel!

Such software might be offered as free and unsupported, at media cost, or as a chargeable product. Whichever way, it needs to have at least some documentation, (i.e., overview and glossary) but it does not have to be a professional package.

We have an initial enquiry from a user who needs a 3-term controller program for servo control, and some process mathematics for numerical filtering and linear conversion. As he said to us, "surely someone has done this before and written it up enough to be useful?". So can you help? If you're offering something free, perhaps we can do a trade for something you would like.

If people are interested in application exchanging we would be happy to act as a 'node' for making contacts. And where someone has some software that has a marketable value, we are interested in helping to create and promote viable packages. We'll not make any firmer plans or suggestions until we hear from you!

Nic Vine
Director
COMSOL
Treway House
Hanworth Lane
Chertsey, Surrey KT16 9LA

Benchmark Battles

Dear Fig:

I believe that the primary consideration of an implementation be fluency of use, and not speed or size except when specific problems arise. But after reading the "Product Review" in FORTH Dimensions III/1, page 11 and seeing some benchmarks, I couldn't resist trying the same on my own home-brew implementation: 4mHz Z-80, S-100 bus (one wait state on all memory ref's). These are the results I got, plus another column correcting for my slower clock (but not for the wait state). I guess I designed for speed.

Just want to stick up for the ol' Z-80. If other people can brag about how compact their implementations are, can't I brag about how fast mine is?

	Timin	**Duncan**
LOOPTEST	2.3	2.9
-TEST	5.9	7.4
*TEST	44.0	54.9
/TEST	74.3	88.6

	Bonadio	**4*61**
LOOPTEST	1.7	1.1
-TEST	6.8	4.5
*TEST	17.5	11.7
/TEST	29.4	19.6

Note

All times in seconds. Each test involves 32767 iterations.

No, I don't use any special hardware. Just the normal Z-80 instruction set. That mulitply threw me off when I first timed it, but the cycles add up about right. I just can't figure out why everyone else is so slow.

I don't have mass storage. That's why I skipped the last two benchmarks. I store everything in EPROMs. Much faster than those clumsy mechanical devices.

Allan Bonadio
1521 Acton St.
Berkeley, CA 94702

Editor's Note:

Here is the code for the benchmarks published in Volume III, No. 1:

```
: LOOPTEST
  7FFF 0 DO LOOP ;
: -TEST
  7FFF 0 DO I DUP  -  DROP LOOP ;
: *TEST
  7FFF 0 DO I DUP  *  DROP LOOP ;
: /TEST
  7FFF 0 DO 7FFF  I  /  DROP LOOP ;
```

To "G" or not to "G"

Dear Fig,

I would like to comment on the "Starting FORTH Editor." The "M" command is bad for reasons of safety and philosophy. It takes a line from the current screen, and puts it "out there" somewhere. If it goes to the wrong place (these things happen), good luck finding it.

A far better alternative is the inverse command, which I call "G" for "get." G takes the same parameters as M (block/line-) and gets a line onto the current screen. I believe that only the screen being edited should change. M violates this rule, G does not.

One further point: G inserts the new line at the current line, not under it. This allows you to alter line 0, which M cannot.

The next extension is BRING , which gets several lines. It takes (block/line/count-). I find G and BRING extremely useful. Comments are solicited.

Mike Perry

I agree! G is more satisfying from the user's point of view. With M, I find myself checking back and forth between the source and destination blocks repeatedly.

The problem of copying a line onto line zero with "M" reminds me of the same problem one has with "U" (also in the "Starting FORTH" editor). I'd like to point out a simple way to "push" a line onto line zero, moving the current line zero and everything else down:

```
0 T U This will be the new line zero
0 T X U
```

The second phrase swaps lines zero and one.--ed.

FORTH in its Own Write

Dear Fig,

The two paragraphs below appeared in an article in BYTE Magazine on pg. 109 of the August 1980 issue. When it first appeared, I agreed with what it was saying but did not feel the need to point it out to others. Now, however, I think that it's time to remind all of us about FORTH and what it isn't. Clearly it isn't any other language.

> The most important criticism of FORTH is that its source programs are difficult to read. Some of this impression results from unfamiliarity with a language different from others in common use. However, much of it results from its historical development in systems work and in read-only-memory-based machine control, where very tight programming that sacrifices clarity for memory economy can be justified. Today's trend is strongly toward adequate commenting and design for readability.
>
> FORTH benefits most from a new, different programming style; techniques blindly carried over from other environments can produce cumbersome results.

It still eludes me as to why people insist on building things into FORTH which are "imports" from other language structures and that in most places do not have any logical place in FORTH. Surely they would not be used by a good FORTH programmer. Take as a simple example spacings. FORTH does not impose indentation or strict spacing requirements as do some other constructs, so why do people insist on indenting? I disagree that this contributes to the readability of the language as FORTH is one of the most terse constructs in existence. One might say that a first attempt to improve the readability of FORTH should center around removing the cryptological do-dads that are used. For instance, "@" should be renamed "FETCH". Likewise, " ! " should be renamed "STORE" and "." changed to "PRINT".

Obviously this is absurd and so is the notion of indentation and other pseudo spacing requirements that some say contribute to "good programming style." Good programming style is writing clear, concise, fast code that does simple things and then using that and other code to construct more complex definitions. This is the premise upon which FORTH was based. I have seen readable code that was sloppily written, too big for the job that it attempted to accomplish and in a single word was abominable. However, it "looked neat and clean."

When the FORTH 79 standard was released I applauded. We are all aware of the small ambiguities and possible deficiencies in the standard. However, the standards team must be commended merely because they exist and they at least attempted to create a standard of some kind. Why then don't people write in standard code? It aggravates me to see code in your journal prefixed or post-addended by a phrase similar to "all you need to do to bring this code up to the standard is..." Why not write standard code in the first place?

This letter is purposely provocative and I sincerely hope that you decide to publish it. Through it I hope to force a reevaluation of the way some individuals look at FORTH. Some of us still think that FORTH is elegant because of its simplicity. It is unfortunate that many refuse to see FORTH as the beautiful language that it is, but see it only as another language that they'd like to resemble.

J.T. Currie, Jr.
Virginia Polytechnic Institute
Blacksburg, VA 24061

Well-expressed, on both points! Regarding the use of the 79-Standard, see our "New Policy" at the front of this issue.--ed.

Minnesota Chapter

Dear fig,

Greetings from the Frozen Wasteland!

This letter is to inform you of the formation of a Minnesota chapter of the FORTH Interest Group. We have had two meetings so far, with attendances of twelve and sixteen respectively. We plan to be meeting once a month. Anyone who is interested should get in contact with us first at the above address.

We hope to start some kind of newsletter in the near future. I've heard that it's possible to get copies of program listings and other handouts which have appeared at Northern California meetings. Could you please let us know how we go about getting copies? I have enclosed a SASE for you to respond.

One of our members is running a Conference Tree (a Flagship for The Commui-Tree Group) which we hope to use for interchange of ideas, programs, etc. outside the general meeting, and to complement the newsletter. The phone number for that Tree is (612) 227-0307. The FORTH branch is very sparse right now, however, since we are just getting off the ground.

We are also contacting local computer groups about jointly sponsoring FORTH tutorials for specific machines, and providing a public-domain, turn-key FORTH system that will turn on their machines. We currently have such software for the Apple II, SYM-1, are close on an Osborne-1, close on an OSI, and are seeking out a TRS-80 version.

Well, that's our plans for the next few months. We would appreciate your current mailing list of Minnesota residents (55xxx and 56xxx zip codes, I believe).

Hope to hear from you soon!

Mark Abbott
Fred Olson
Co-founders of MNfig

Happy to hear about your new chapter! Your mailing list is on its way. And yes, handouts from the Northern California Chapter meetings are available. Here's how to obtain them:

John Cassady of the Northern California chapter has agreed to serve as a clearinghouse. The Secretary of any FIG Chapter can mail, each month, handouts from his own Chapter's meetings to Mr. Cassady. In return, John will send back one set of all handouts he receives each month, including those from the Northern California meetings. Even if a local Chapter has no handouts, the Secretary must sent at least a postcard to indicate the Chapter's continued interest. The local Chapter's Secretary will make the necessary copies to distribute to members of that Chapter.

So, let's see those handouts from all the Chapters! Write to:

John Cassady
339 15th Street
Oakland, CA 94612

Brain-System

Dear fig,

The special FORTH issue of Dr. Dobb's Journal made a deep impression on me and on my son. My son is since 12 years a system programmer and knows more than a dozen computer programming languages. I am a logician and engineer, code designer and the developer of the only existing proto-model of Interdisciplinary Unified Science and its computer-compatible language, the UNICODE.

Thus, I represent a radically different path of scientific development--disregarded by many because it does not promise immediate financial returns.

My approach is centered on a new and far more encompassing system-idea of the temporary name "brain-system" having a physical-hetero-categorical genetically ordered sequence of models of logic. This sequence has a specific case for present-day formal logic and a corresponding simplified variant of the system-idea: this is the system-idea of the digital computer.

UNICODE is the first specific brain-system programming language. It is a content oriented language, it has powerful semantics and register-techniques. It has "words" which are at the same time total programs for the generation of the invars and "content" the term intends to communicate.

I think to study UNICODE will lead to unsuspected breakthrough in the development of programming, especially if thinking has been made elastic and modular by studying FORTH.

I would like to receive the private addresses of a few creative FORTH fans. In the hope of your early reply, I remain...

Prof. Dipl. Ing. D.L. Szekely
P.O. Box 1364
91013 Jerusalem, Israel
December 1981

Anyone follow that?--ed.

TECHNOTES

ENCLOSE Correction for 6502

Andy Biggs
41, Lode Way
Haddenham
Ely, Cambs
CB6 3UL
England

On converting my 6502 fig-FORTH (V1.1) to work with 256 byte disc sectors, I discovered (after many system hang-ups) that WFR's 'ENCLOSE' primitive is not guaranteed to work with disc sector sizes greater than or equal to 256 bytes in size.

In his 'ENCLOSE,' Bill uses the 6502 Y register to index through the input text stream, but this register is only 8 bits, so if the text stream contains a block of delimiter characters, e.g., 'space' bigger than 256, it will loop forever, as I found to my cost!

When will this occur? Never from the terminal input buffer, which is only 80 characters long.

With a disc sector size of 256 or bigger, if you have an entire sector of spaces in a load screen, then the load will hang up on this chunk of spaces.

or...

If your sector size is bigger than 256, then any chunk of spaces 256 or bigger will hang it.

I encountered this because I decided to emulate John James' method used on the PDP-11 version, where 'R/W' handles 1K every time, so as far as BLOCK, BUFFER, and ENCLOSE are concerned, the disc block is 1024 bytes, and compiling hung up on any text gap bigger than 256 bytes!

Anyway, I ENCLOSE (ha ha) a revised version of the ENCLOSE primitive which I am now using, which has full 16 bit indexing. I'm sure some assembly language programmer could produce a neater version, but at least I know that this one works.

Keep up the good work.

By the way, I'm willing to act as a fig software exchange/library in the UK, unless there is someone already doing it?

```
          'ENCLOSE' PRIMITIVE FOR 65Ø2 WITH 16-BIT INDEXING

          THE 'Y' REGISTER FORMS THE LOW INDEX BYTE
          STACK LOCATION $1,X FORMS THE HIGH INDEX BYTE
          THE BASE ADDRESS HELD IN $N+2 , $N+3 IS ALSO AFFECTED

          .BYTE    $87,'ENCLOSE'
          .WORD    L243
          .WORD    *+2

          LDA      #$2
          JSR      $SETUP
          TXA
          SEC
          SBC      #$8
          TAX
          STY      $3,X
          STY      $1,X                    ; INITIALISE AS BEFORE
                                           ; SETTING HI INDEX = Ø
          DEY
          DEC      $N+3
          DEC      $1,X                    ; PRIME THESE VARIABLES FOR LOOP
L313      INY
          BNE      XXX1
          INC      $N+3                    ; INCREMENT HI ADDRESS
          INC      $1,X                    ;       AND HI INDEX
XXX1      LDA      ($N+2),Y                ; GET CHARACTER FROM INPUT STREAM
          CMP      $N                      ;     IS IT DELIMITER ?
          BEQ      L313                    ;     LOOP IF TRUE

          STY      $4,X                    ; NON-DELIMITER SO PUT FIRST
          LDA      $1,X                    ; RESULT ON THE STACK
          STA      $5,X

L318      LDA      ($N+2),Y                ; GET CHARACTER AGAIN
          BNE      L327                    ; BRANCH IF NOT A NULL

          STY      $2,X
          STY      $Ø,X                    ; TIDY UP RESULTS FOR 'NULL' EXIT
          LDA      $1,X
          STA      $3,X
          TYA
          CMP      $4,X                    ; IF FIRST AND LAST INDEXES ARE EQUAL
          BNE      L326
          LDA      $1,X
          CMP      $5,X                    ;       THEN
          BNE      L326
          INC      $2,X                    ;           INCREMENT THIS RESULT
          BNE      L326
          INC      $3,X
L326      JMP      NEXT

L327      PHA                              ; SAVE CHARACTER
          STY      $2,X
          LDA      $1,X                    ;       SAVE CURRENT INDEX AS OFFSET TO
          STA      $3,X                    ;       FIRST DELIMITER AFTER TEXT
          INY
          BNE      XXX5
          INC      $1,X                    ; INCREMENT INDEX
          INC      $N+3                    ; AND HI ADDRESS
XXX5      PLA                              ; RECOVER CHARACTER




          CMP      $N                      ; IF NOT DELIMITER
          BNE      L318                    ;    THEN LOOP
          STY      $Ø,X                    ;    ELSE EXIT
          JMP      NEXT
```

TRANSIENT DEFINITIONS

Phillip Wasson

Editor's Note: This article appeared in the last issue, but, unfortunately, without the source code. Here is the article as it should have appeared. Our apologies.

These utiliites allow you to have temporary definition (such as compiler words: CASE, OF ENDOF, ENDCASE, GODO, etc.) in the dictionary during compilation and then remove them after compilation. The word TRANSIENT moves the dictionary pointer to the "transient area" which must be above the end of the current dictionary. The temporary definitions are then compiled into this area. Next, the word PERMANENT restores the dictionary to its normal location. Now the application program is compiled and the temporary definitions are removed with the word DISPOSE. DISPOSE will take a few seconds because it goes through every link (including vocabulary links) and patches them to bypass all words above the dictionary pointer.

NOTE: These words are written in MicroMotion's FORTH-79 but some non-79-Standard words are used. The non-Standard words have the fig-FORTH definitions.

```
FIRST 1000 - CONSTANT TAREA  ( Transient area address )
VARIABLE TP  TAREA TP !      ( Transient pointer )
: TRANSIENT ( --- ADDR )
    HERE TP @ DP ! ;
: PERMANENT ( ADDR --- )
    HERE TP ! DP ! ;
: DISPOSE ( --- )
    TAREA TP !  VOC-LINK
   BEGIN DUP
    BEGIN @ DUP TAREA U< UNTIL DUP ROT ! DUP 0=
   UNTIL DROP  VOC-LINK @
   BEGIN DUP 4 -
    BEGIN DUP
     BEGIN PFA LFA @ DUP TAREA U<
     UNTIL DUP ROT PFA LFA !  DUP 0=
    UNTIL DROP @ DUP 0=
   UNTIL DROP  [COMPILE FORTH DEFINITIONS ;

 ( Example )
TRANSIENT
: CASE     ... ;
: OF       ... ;
: ENDOF    ... ;
: ENDCASE ... ;
PERMANENT
: DEMO1
     ... CASE
      ... OF ... ENDOF
      ... OF ... ENDOF
         ENDCASE ;

TRANSIENT
: EQUATE ( N --- )
   CREATE , IMMEDIATE
   DOES> @ STATE @
    IF [COMPILE LITERAL THEN ;
7 EQUATE SOME-LONG-WORD-NAME
PERMANENT
: DEMO2                          ( SOME-LONG-WORD-NAME is compiled)
   SOME-LONG-WORD-NAME . ;       ( as a literal )

DISPOSE        ( Removes the words EQUATE, SOME-LONG-WORD-NAME, )
               ( CASE, OF, ENDOF, and  ENDCASE from the )
               ( dictionary. )
DEMO2 7 OK     ( Test DEMO2, it prints a seven. )
```

NOVA bugs

John K. Gotwals
Computer Technology Department
South Campus Courts C
Purdue University
W. Lafayette, IN 47907

I have just finished installing fig-FORTH on my NOVA 1200, using the listing I received from fig. Instead of running it standalone, as the fig listing does, I run it as a task under RDOS Rev. 5.00.

So far I have found four bugs or omissions in the listing. They are as follows:

Page 10 of the listing - EMIT does not increment OUT.

[COMPILE] does not work properly. It can be fixed by removing CFA, from line 07 on page 42 of the listing.

VOCABULARY does not work properly. This can be fixed by adding CFA between AT and COMMA on line 53 of page 44.

(FLUSH) can not be accessed until a missing <51> is inserted after FLUSH on line 13 of page 52.

After installing fig FORTH, I entered the CYBOS editor from the keyboard and used this editor to boot the fig editor listed in the installation manual. After this experience, I am somewhat pessimistic about FORTH's portability between word and byte addressing machines. I had to make quite a few changes before the fig editor would run. Some examples:

BLANKS expects a word address and word count.

COUNT expects a word address and returns a byte address.

HOLD and PAD both return word addresses.

If any RDOS NOVA users would like a copy of my "fig-FORTH," they should feel free to contact me.

FORTH Standards Corner

Robert L. Smith

DO, LOOP, and +LOOP

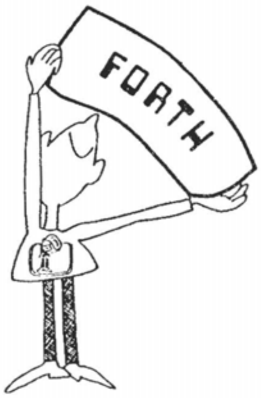
There have been some complaints about the way that +LOOP is defined in the FORTH-79 Standard. The first obvious problem is that the Standard does not define the action to be taken when the increment n is equal to zero. Presumably that was either an oversight, or a typographical error. The most likely correction is to treat the n=0 case the same as n>0, since the arithmetic is defined to be two's complement, and for that arithmetic, the sign of 0 may be considered to be positive. I am aware of other possibilities, but they seem to be fairly difficult to implement or explain.

The second point that is mentioned is that the parameter range seems to have a strange asymmetry. When a positive increment is used, the DO-LOOP index I may not reach the specified limit. However when a negative increment is used, the index I may be equal to the specified increment. Users of fig-FORTH systems have pointed out that the fig +LOOP is symmetric in the sense that for either negative or positive increments the limit value is never reached. One may consider that the Standard version terminates when the boundary between the limit n and n-1 is crossed, whether the increment is positive or negative.

Finally it has been noted that the Standard LOOP and +LOOP depend on signed arithmetic. Many, but not all, FORTHs use a modular or circular arithmetic on DO-LOOPs, allowing the index I to directly address memory. The use of I to address memory in a Standard LOOP may result in a non-transportable program unless a certain amount of care is taken. The Standard version is easier to define than one involving circular arithmetic. Note also that the Standard version allows approximately twice the range of most circular loops (such as in fig-FORTH).

The best suggestions for new looping methods can be found in a paper given by Robert Berkey at the recent FORML Conference. The paper is entitled "A Generalized FORTH Looping Structure." I recommend that readers interested in the topic get a copy of this paper and implement his suggested words. I wqould like to slightly modify his results for the current discussion. Berkey essentially shows a technique for looping in which the increment for +LOOP may alternate between positive and negative values without necessarily terminating the loop. Modular arithmetic is used so that either signed or unsigned use of the index I may be employed. The increment may be any value. The terminating condition is when the boundary between n and n-1 (actually n+1 in Berkey's paper) is crossed dynamically. The implementation appears to be even more efficient than that described by Brodie and Sanderson ("Division, Relations, and Loops," Rochester Conference, 1981). The only apparent disadvantage of the implementation is that the index is computed by addition or subtraction. A novel feature of Berkey's implementation is that when the word LEAVE is executed, the loop is terminated at that point (i.e., LEAVE actually leaves). Berkey also suggests that for normal positive incrementing loops that the index range should include the upper limit, in a manner more consistent with other languages as well as typical use in the fig-FORTH INDEX. Finally, he suggests a construct so that a loop may be skipped entirely if a counting parameter is zero.

The work discussed above is of potential interest to future directions in FORTH. It shows that FORTH is still evolving, even though it cannot effect the current Standard.



Position Wanted

I am looking for a software engineering position with another company that uses FORTH. I would like to work for a firm using FORTH to develop state-of-the-art systems software; specifically, a FORTH-based development and oeprating system environment to compete head on with UNIX.

Brent Hoffman
13533 37th N.E.
Seattle, WA 98125
(206) 363-0642

9900 Trace

Heinz F. Lenk
Loewensteiner Ring 17
6501 Woerrstadt
Germany

I have had some trouble getting my 9900 FORTH running.

To ease the finding of errors I wrote a program to display all important vectors (IP, W, CODE, R, SP) and the first 7 stack contents. Even the stack's growing is visible.

I would like to contribute it to you, so you can offer it to all 9900 users with a 100M or similar board.

It was a great luck for me that I did not need the addresses >37C and >37E, and could use it for a branch to the STATUS program. This program is switched off by the code HEX 455 384 ! and switched on by HEX 457 384 !.

The program list contains the routines for terminal input and output, too.

I hope I can help some people with my program.

```
           *      fig-FORTH 9900
           * SYSTEM DEPENDEND CODE FOR 990/100M BOARD
           * FOLLOWING PROGRAMS SUPPORT THE 'KEY' AND
           * 'EMIT' INSTRUCTIONS.
           *
           * TO EASE ERROR FINDING THERE IS A PROGRAM
           * TO PRINT OUT THE POINTERS AND THE FIRST
           * SEVEN STACK CONTENTS
           * HEINZ LENK, LOEWENSTEINER RING 17, 6501 WEORRSTDT
           * GERMANY.                                  12/06/81
          *************************************************
                        AORG >44
 44  FF00               DATA >FF00      XOP 1 VECTOR WP
 46    80               DATA RENTRY     XOP 1 VECTOR PC
 48  FF00               DATA >FF00      XOP 2 VECTOR WP
 4A    8E               DATA EMITSP     XOP 2 VEXTOR PC
 4C  FF20               DATA >FF20      XOP 3 VECTOR WP
 4E    C8               DATA MENTR      XOP 3 VECTOR PC
 50  FF20               DATA >FF20      XOP 4 VECTOR WP
 52    D2               DATA WENTRY     XOP 4 VECTOR PC
           ************************************************
           * READ DATA TO STACK POINTED BY R8
           * CALL WITH  XOP *R8,1
                        AORG >80
 80   20C RENTRY        LI   R12,>80      SET CRUBASE
 82    80
 84  1F15               TB   21           RECEIVER BUF. FULL
 86  16FC               JNE  RENTRY
 88  35DB               STCR *R11,7        FETCH ASCII
 8A  1E12               SBZ  18              RECEIVER INTR DSBL
 8C   380               RTWP
           ************************************************
           * WRITE A CHARACTER TO TERMINAL 'EMIT'
           * CALL WITH XOP *R8,2
           *
 8E   20C EMITSP        LI   12,>80          SET CRUBASE
 90    80
 92  1D10               SBO  16              RTS ON
 94  1F16 T22           TB   22             TRANSMIT BUF. EMPTY?
 96  16FE               JNE  T22
 98  1F1B T27           TB   27             DSR ?
 9A  16FE               JNE  T27
 9C  321B               LDCR *R11,8         OUTPUT
 9E  1F16 TE22          TB   22              TRANSMIT BUF. EMPTY?
 A0  16FE               JNE  TE22
 A2  1F17               TB   23              TRANSMIT SHIFT EMPTY?
 A4  16FC               JNE  TE22
 A6   380               RTWP
           *************************************************
 A8  0A0D S1            DATA >0A0D          CR,LF
 AA                     TEXT "IP="
 AE       S2            TEXT "  W="
 B4       S3            TEXT "  CODE="
 BC       S4            TEXT "  R="
 C2       S5            TEXT "  SP="
```

```
            ***********************************************
            * SUBPROGRAM TO OUTPUT A STRING TERMINATED BY >00
            * CALL WITH  XOP §ADRESS,3
            *
 C8  D0BB MENTR     MOVB *R11+,R2   FETCH BYTE
 CA  1302           JEQ  MEXIT       EXIT IF ZERO
 CC  C282           XOP  R2,2        PRINT ASCII CHAR.
 CE  10FC           JMP  MENTR
 D0   380 MEXIT     RTWP
            ***********************************************
            * SUBPROGRAM TO OUTPUT A HEX WORD
            * CALL      XOP  SOURCE,4
            *
 D2   201 WENTRY    LI   R1,4       COUNT
 D4     4
 D6  C2EB           MOV  *R11,R11    FETCH WORD
 D8   B4B           SRC  R11,4       ALIGNMENT
 DA  C08B WNEXT     MOV  R11,R2      COPY
 DC   242           ANDI R2,>0F00   MASK OUT
 DE   F00
 E0   282           CI   2,>900     NUMBER?
 E2   900
 E4  1202           JLE  NUM
 E6   222           AI   R2,>700    ADJUST LETTER
 E8   700
 EA   222 NUM       AI   R2,>3000   ADJUST ASCII
 EC  3000
 EE  2C82           XOP  R2,2        OUTPUT
 F0   BCB           SRC  R11,12      SHIFT
 F2   601           DEC  R1          COUNT-1
 F4  16F2           JNE  WNEXT       ZERO ?
 F6   380           RTWP            EXIT
            *****************************************
            * PRINT STATUS PROGRAM
            * USED FOR DEBUG DURING SET UP
 F8  1000 STATUS    NOP             SPARE
 FA  2CE0           XOP  §>A8,3      MSG ZIP
 FC    A8
 FE  C009           MOV  R9,R0       COPY
100   640           DECT  R0         OLD ZIP
102  2D00           XOP  R0,4        OUTPUT ZIP
104  2CE0           XOP  §>AE,3      MSG ZW
106    AE
108  C00B           MOV  R11,R0
10A   640           DECT R0
10C  2D00           XOP  R0,4
10E  2CE0           XOP  §>B4,3     MSG CODE
110    B4
112  2D05           XOP  R5,4
114  2CE0           XOP  §>BC,3     MSG ZR
116    BC
118  2D0A           XOP  R10,4
11A  2CE0           XOP  §>C2       MSG ZSP
11C    C2
11E  2D08           XOP  R8,4
120  1000           NOP
            * OUTPUT 7 STACKS
122   201           LI   R1,7    COUNT
124     7
126  C0E0           MOV  §>31A,R3   FETCH STACK START STAX
128   31A
12A   202           LI   R2,>2000    SPACE
12C  2000
12E  2CC2 STOUT     XOP  R2,3       MSG BLANC
130  2D13           XOP  *R3,4      PRINT HEX
132  8203           C    R3,R8     CURRENT STACK POINTER?
134  1303           JEQ  STEXIT     JUMP IF EQUAL
136   643           DECT R3        NEXT STACK
138   601           DEC  R1         COUNT 1
13A  16F9           JNE  STOUT      REPEAT UNTIL ZERO
13C   455 STEXIT    B    *ZTEMP1       RESUME WORK BY *R5
            *
            ************************************************
            * THE ORIGINAL DYNAMIC RAM ALLOCATION PROGRAM DOES
            * NOT WORK WITH AN UNTERMINATED DATABUS.
            * THIS 4 LINES SOLVE THE PROBLEM.
                    AORG >34A
34A   4D5           CLR  *R5        CLEAR RAM ADDRESS
34C  C555           MOV  *R5,*R5   DUMMY
34E  13FB           JEQ  SEARCH    JUMP BACK IF ZERO
350  1000           NOP
            *
            *********************************************
            * THE INNER INTEPRETER IS CHANGED TO PRINT ALL
            * POINTERS (IP,W,CODEBODY,R,SP) AND STACK.
            * THE STATUS IS SWITCHED ON BY      HEX 457 384 !
            * THE STATUS IS SWITCHED OFF BY     HEX 455 384 !
                    AORG >37C
37C   207           LI   R7,>F8       PC OF STATUS
37E    F8
380  C2F9           MOV  *ZIP+,ZW
382  C17B           MOV  *ZW+,ZTEMP   POINT TO BODY
384   457           B    *R7        BRANCH TO STATUS
```

A TECHNIQUES TUTORIAL: EXECUTION VECTORS

Henry Laxen
Laxen & Harris Inc.
24301 Southland Drive
Hayward, CA 94545

This month, we continue our exploration of FORTH programming techniques by taking a look at a concept known as Execution Vectors. This is really a fancy name for very simple concept, namely using a variable to hold a pointer to a routine that is to be executed later.

It is only fair to warn you that the dialect of FORTH that I am using is the one discussed in Starting FORTH by Leo Brodie. It has several differences from figFORTH, not the least of which is the fact that in figFORTH EXECUTE operates on code field addresses (cfa's), while in Starting FORTH EXECUTE operates on parameter field addresses (pfa's). This may not seem like a big deal, but if you have ever fed EXECUTE a pfa when it was expecting a cfa, you have undoubtedly remembered the result. Anyway, my EXECUTE uses pfa's. Its function is to perform or EXECUTE the word that this pfa points to. An example will clear this up. Suppose we have the following:

```
: GREET ." HELLO, HOW ARE YOU" ;
' GREET ( LEAVE THE PFA OF
             GREET ON THE STACK )
  EXECUTE ( AND NOW PERFORM IT )
```

the result is:

```
HELLO, HOW ARE YOU
```

which is the same result as just typing GREET.

The above may not seem too significant, but the implications are tremendous. Consider the following examples:

```
VARIABLE 'EMIT

: EMIT    ( CHAR --- )
   'EMIT @ EXECUTE ;

' (EMIT) 'EMIT !
```

I assume that (EMIT) is a routine which takes a character from the stack and sends it to the terminal. By defining EMIT to use 'EMIT as an execution vector, we now have the ability to redirect the output of FORTH in any manner we choose. For example, suppose we want all control characters that are sent to the screen to be prefixed with a caret. We could do the following:

```
: CONTROL-EMIT    ( CHAR --- )
   DUP 32 ( BLANK ) < IF     ( Control Char? )
      94 ( ^ ) (EMIT)        ( Yes, emit an ^ )
      64 ( ASCII A - 1 ) +   ( and convert it )
   THEN
   (EMIT) ;

' CONTROL-EMIT 'EMIT !
```

Now all regular characters will fail the test, since they will be larger than blanks; however, control characters will succeed and will be incremented by 64, making them displayable.

There are several other FORTH words that have proven useful to vector. Some of these include:

KEY input from keyboard primitive

CREATE change header structures

LOAD useful for many utilities

R/W disk i/o primitive

For example, if LOAD were vectored, then by redefining it to print a screen instead of loading it, you could write a print utility which prints screens in load order by LOADing a load screen and redefining LOAD to print. CREATE could be changed to add the screen number of each definition to the dictionary header so that it could later be retrieved with VIEW or the equivalent. KEY may be changed to get its characters from a file somewhere instead of the keyboard. In short, there are a thousand and one uses for Execution Vectors.

But be careful, I may have opened Pandora's box with the above selling job. There is a price to be paid for execution vectors, and that is complexity, the archenemy of reliability. Every word that you decide to vector at least doubles the complexity of the FORTH system you are running, since it introduces at least two or more states that the system can be in. You must now also know what the version is of each execution vector you are using. If you have 3 different EMITs and 2 different KEYs and 3 different LOADs, you have a total of 18 different states that the system can be in just on these vectors alone. So use vectors sparingly, otherwise you will lose control of the complexity very very quickly.

Having decided to use execution vectors, we're now faced with different approaches towards implementing them. The one described above works, and is used by many people, but it has one unfortunate property, namely the need to name a variable which is basically overhead. Here is another way to accomplish the same thing without having to define a variable. Consider the following:

```
: DIE      ( --- )
   1 ABORT" THIS WOULD HAVE CRASHED!" ;

: EXECUTE:
   CREATE     ( --- )
      [']  DIE ,
   DOES>      ( --- )
      @ EXECUTE ;

: IS           ( PFA --- )
   ' ! ;
```

DIE is used to send an error message to the terminal and reset the FORTH system into a clean state. EXECUTE: is a defining word which initializes itself to DIE, but hopefully will be changed later by the user. Words defined with EXECUTE: can be changed with IS as follows:

```
EXECUTE: EMIT

' (EMIT) IS EMIT      (or perhaps)
' CONTROL-EMIT IS EMIT
```

What EXECUTE: has done is combined the variable name with the Execution Vector name into one name. IS is used as a convenience, so that the user can forget the internal structure of words defined by EXECUTE:. Also it provides an extremely readable way of redefining Execution Vectors. Notice that as defined, IS may only be used during interpretation. I leave it as an exercise for the reader to define an IS that may be compiled within : definitions.

Another approach to redefining execution vectors is via the word ASSIGN. It could be defined as follows:

```
: (ASSIGN)          ( CFA --- )
   R> 2+ SWAP ! ;

: ASSIGN            ( --- )
   COMPILE (ASSIGN)
   [ ' : CFA @ ] LITERAL ,     ; IMMEDIATE
```

It would be used as follows:

```
: UPPER-ONLY         ( --- )
   ['] EMIT ASSIGN
     DUP 96 ( ASCII a-1 ) > IF
        DUP 123 ( ASCII z+1 ) < IF
           32 -
        THEN
     THEN
     (EMIT) ( AS ALWAYS ) ;
```

When UPPER-ONLY is executed, EMIT is redefined to execute the code following the ASSIGN, which will convert all lower case characters to upper case, and send them to the terminal. Note that unlike IS, ASSIGN may only be used within : definitions.

That's all for now, good luck, and may the FORTH be with you.

CHARLES MOORE'S BASIC COMPILER REVISITED

Michael Perry

In this paper I will discuss several interesting features of the "BASIC Compiler in FORTH" by Charles Moore (1981 FORML Proceedings).

Why is a BASIC compiler interesting? There are a number of reasons. Foremost of them is that BASIC is in many ways typical of a variety of popular languages, particularly FORTRAN, PASCAL, and ADA. Conspicuous features of these languages are algebraic notation, lack of access to the underlying hardware, poor input and output facilities, and non-extensibility. FORTRAN and BASIC also suffer from poor structuring due to the extensive use of GOTO. These languages all tend to be best at solving equations. Other prominent features of BASIC are it s use of statement numbers as labels, low speed, and its use of a few complicated functions (e.g., PRINT) rather than many simple ones.

Why is it slow? BASIC interpreters usually convert source code statements to an intermediate form, where keywords become tokens. The token interpreter is slow because tokens must be deciphered (translated into actions) at run time. This BASIC to FORTH compiler produces code which runs unusually fast. This is because it produces FORTH object code, i.e., sequences of addresses of code routines.

You should look at the example programs (blocks 80-82) before reading the text. You will notice that each BASIC program becomes a FORTH word named RUN. It is executed by typing its name, i.e., RUN. This is how BASIC usually works; you type RUN to execute the program. It serves to demonstrate that from FORTH's point of view, BASIC only knows one "word," RUN. Is it not more useful and flexible to let routines have any name, and to be able to execute any of them by typing its name? Yes, and that is a key feature of FORTH.

How It Works

I will refrain from commenting on the intrinsic value of a BASIC compiler; that has already been covered well in Moore's paper. The principal features I will discuss are the handling of operator precedence, variables in algebraic equations, and the use of the FORTH compiler. The most important part of this BASIC compiler is its ability to convert algebraic (infix) source code to reverse polish (postfix) object code.

A BASIC program is compiled inside the colon definition of a word named RUN. This means that the FORTH system is in its compile state, and any words to be executed during compilation must be immediate. This use of the FORTH compiler was perhaps my greatest lesson from studying this BASIC compiler. The ordinary FORTH compiler is far more versatile than I had realized. If I had written this compiler, it would doubtless have run in the execution state and would have been far more complicated as a result.

Let's look at an example. The BASIC statement
10 LET X = A + B
will be compiled into object code equivalent to the FORTH expression
X A @ B @ + SWAP !
where X, A, and B are variables. One of the variables (X) returns an address, the rest return values (with a fetch). The add is compiled after the fetches of the values to be added. The equals becomes the " SWAP ! " at the end. Because the source code (in BASIC) is in algebraic notation, and the (FORTH) object code is in reverse polish order, some way is needed to change the order of operations when compiling the BASIC program. The mechanism which controls the compilation order is based on the idea of operator precedence, which means that some operators are assigned higher priority than others.

PRECEDENCE

The idea of operator precedence is a prominent feature of most computer languages (FORTH is a notable exception). Operations are not necessarily performed in the order you specify. An example will help. The equation X = 5 + 7 * 2 could mean either X = (5 + 7) * 2 or X = 5 + (7 * 2), usually the latter. In FORTH this would be 7 2 * 5 + X ! where the order is explicit. In algebraic languages some method is needed to clarify the order of evaluation of operators in expressions. That is what precedence does. Each operation is assigned a precedence level. Operations with higher precedence are performed earlier.

During compilation of the BASIC program (the FORTH word named RUN) the compilation of many words is deferred. This allows the order of words to differ between the source code and the object code. Take '+' as an example. To defer compilation of '+' a new word is created which is immediate (and so executes at compile time). When this new word is executed, it leaves the address of '+' on the stack, and on top it leaves the precedence value of '+'. The defining word PRECEDENCE creates the new word as follows: " 2 PRECEDENCE + " . This creates a new, immediate word named '+', which will leave the address of the old word '+' under the value 2.

The word which decides how long to defer compilation is DEFER. DEFER looks at two pairs of numbers on the stack. Each pair consists of an address and a precedence value. If the precedence of the top pair is larger than that of the lower, DEFER does nothing. If the top precedence is less than or equal to the one below, the address part of the lower pair is compiled, and its precedence is discarded. DEFER will continue to compile until the upper precedence is larger than the lower.

So how do you get started? Essentially, most BASIC keywords (such as LET) execute START wqhich leaves 'NOTHING 0 on the stack, where 'NOTHING is the address of a do nothing routine and 0 is its precedence. This pair will remain on the stack during the compilation of that statement, because everything has higher than zero precedence.

At the end of each line, RPN is executed. It performs a 0 1 DEFER, which forces the compilation of any deferred words, because every operator has a precedence of at least 1. RPN then consumes the 0 and executes NOTHING. Actually, each statement is ended by the start of the next. BASIC keywords such as LET execute STATEMENT, which contains RPN (to finish the previous statement) and START (to begin the next).

BRANCHING

Three new branching primitives are used. They are compiled by various higher level words. JUMP is used by GOTO. SKIP and JUMP are used by IF-THEN. JUMP is compiled followed by an absolute address. When executed it simply loads that address into the IP (virtual machine instruction pointer). When SKIP executes, it takes a boolean off the stack. If true it adds 4 to the IP, skipping (usually) the following JUMP.

(NEXT) is used for FOR-NEXT loops. It is compiled followed by an absolute address. When executed it takes three parameters from the stack: final value of the loop index, step size, and the address of the variable containing the current value of the loop index. It adds the step (plus or minus) to the variable, and loops until the index passes the limit.

Adding GOSUB would require another branching primitive, CALL.

STATEMENT NUMBERS

Each BASIC statement must be preceded by a number. This number acts as a label, allowing branches between lines. In this compiler, the numerical value of the labels does not affect execution order. When a statement number is encountered, it is compiled in line as a literal. The address of LIT is compiled followed by the literal value 10. For example, when the statment "10 REM" is encountered, 10 is compiled as a literal. The keyword REM is immediate, and so is executed. It begins by executing STATEMENT, which, amongst other things, fetches the value of the line number just compiled (10), and enters it into the statement number table (#S) along with the address (HERE) of the start of that statement. STATEMENT then de-allocates the space used by the literal 10 (with a -4 ALLOT). It scans the table and resolves any forward references to the new statement. When a forward reference occurs, as in "GOTO 50" before statement 50 is compiled, GOTO compiles 'JUMP 0'. The zero will later be replaced by the address of line 50. The reference is entered into the table with the address to be patched instead of the actual address of statement 50. Additional forward references to the same point will be chained to each other. To indicate that this is a forward reference, the address in the table is negated. This means that BASIC programs must be compiled below 8000H, so that all addresses appear to be positive. Here simplicity was chosen over generality.

VARIABLES

There are two particularly interesting things to notice about variables. They are immediate, and they know which side of an equation they are on. Three types of variables are supported: integers, arrays, and two dimensional arrays. Variables must be declared (defined) before use. The BASIC expressions: LET X = A + B (where X, A, and B are variables) compiles into the following FORTH equivalent:

```
X A @ B @ + SWAP !
```

Notice that when an integer appears on the left of an equals sign, it must compile its address, and when on the right side, its value (address, fetch). Also note that only one can appear on the left, while many can be on the right.

The way this is implemented is surprisingly simple. The variable ADDRESS contains a flag which indicates which side of the equals sign a variable is on. The word LET sets ADDRESS to 1. "INTEGER X" creates a variable named X, which is immediate. When X is executed it compiles its address. X then examines ADDRESS. If it is true (non-zero), X simply makes it zero. If ADDRESS is false, X compiles a @ after the address, thereby rturning the value when the BASIC program is run.

Notice that the equals sign plays no role in this process; everything is done by keywords (e.g., LET) and variables.

Future Directions

Many more features can easily be added to this BASIC compiler. But why bother? A much more fruitful line of endeavor would be to make use of the lessons learned in this compiler to write compilers for other, more useful, languages such as C. A C compiler which is easy to modify and extend, and just as portable as FORTH is, could actually be useful. Another area worthy of effort might be generators for machine code, a common thing for compilers to have.

Conclusions

It is possible to use FORTH to produce portable compilers for other languages. Doing so provides insight into the nature of languages, and the desirability of various approaches to problem solving. Whether the compilers themselves prove useful or not, it is worthwhile to write them.

(screens on following pages)

Transportable Control Structures With Compiler Security

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

This article is an enhancement of the idea presented by Kim Haris at the Rochester FORTH Conference (from the Conference Proceedings, page 97). Basically, the article proposes a wordset of primitives for defining control words such as IF , ELSE , THEN , DO , LOOP , BEGIN , WHILE , REPEAT , UNTIL , AGAIN , CASE , etc. Kim points out that these strucures are either compiling a branch to a location not yet defined (such as IF --> THEN) or back to a location previously defined (BEGIN <-- UNTIL). There are two steps in compiling either kind of branch: marking the first place compiled and then later resolving the branch. This observation leads to four of Kim's words:

>MARK	Marks the source of forward branch and leaves a gap.
>RESOLVE	Resolves forward branch and leaves a gap.
<MARK	Marks destination of backward branch.
<RESOLVE	Resolves backward branch.

I complement Kim at this point for his excellent choice of names. Here's where compiler security comes in.

The word >RESOLVE is filling a gap left by >MARK . If >RESOLVE were to first check to make sure a gap was there (DUP @ 0 ?PAIRS) it would help ensure that the value on the stack was indeed left by >MARK . Likewise, if <RESOLVE made sure that the point where it branches back to does not have a gap (DUP @ NOT 0 ?PAIRS) it would guarantee that it was not answering a >MARK . This method allows some compiler security where it is important not to carry pairs on the stack.

Example:

```
:  >MARK   HERE 0 , ;
:  >RESOLVE   DUP @ 0 ?PAIRS HERE
        SWAP ! ;
:  <MARK   HERE ;
:  <RESOLVE   DUP @ NOT 0 ?PAIRS , ;
:  IF   C, >MARK ;
:  ENDIF   >RESOLVE ;
:  ELSE   C3 IF SWAP ENDIF ;
:  BEGIN   <MARK ;
:  UNTIL   C, <RESOLVE ;
:  AGAIN   C3 UNTIL ;
:  WHILE   IF ;
:  REPEAT   SWAP AGAIN ENDIF ;
```

```
     72
( Charles Moore's BASIC compiler, modified for fig-FORTH )
VOCABULARY ARITHMETIC   ARITHMETIC DEFINITIONS
VOCABULARY LOGIC   VOCABULARY INPUT   FORTH DEFINITIONS

: +LOAD   BLK @ + LOAD ;
: (GET#)   BL WORD HERE NUMBER DROP ;
: (.)   S->D SWAP OVER DABS <# #S SIGN #> ;
0 VARIABLE #S  128 ALLOT
: SCR   0 #S 2+ #S 2! ;
( Precedence )  1 +LOAD 2 +LOAD 3 +LOAD
: [   93 WORD ;  IMMEDIATE
ARITHMETIC DEFINITIONS
( BASIC ) 4 +LOAD 5 +LOAD 6 +LOAD 7 +LOAD
: (   10 #( +! ;  IMMEDIATE
: ;   [ n]   , ;   1 PRECEDENCE ;
FORTH DEFINITIONS
```

```
     73
( Precedence )
0 VARIABLE ADDRESS  0 VARIABLE #(
: )   -10 #( +!  #( @ 0< ABORT" Unmatched )" ;  IMMEDIATE
: DEFER  ( a n a n - a n)  #( @ +
   BEGIN  2OVER SWAP DROP OVER  < NOT
   WHILE   2SWAP DROP  CFA ,  REPEAT ;
: PRECEDENCE  ( n - )  IN @  [COMPILE] ' >R  IN !
   <BUILDS , R> , IMMEDIATE    DOES>  2@  DEFER ;

: RPN  ( n)  0 1 DEFER  2DROP  #( @ OR  ABORT" Syntax" ;
: NOTHING  ;
: START ( - a n)   0 #( !  0 ADDRESS !  ' NOTHING 0
   ARITHMETIC ;  IMMEDIATE
: ?IGNORE   #( @  IF  0 1 DEFER  2DROP  R> DROP  THEN ;
```

```
     74
( Branching - high level )

: JUMP   R> @ >R ;
: SKIP   0= IF  R> 4 + >R  THEN ;
: (NEXT)    ( to \ step \ variable address -- )
   2DUP +! ( add step to var )
   >R 2DUP R> @ SWAP ( t s t v s )
   0< IF  SWAP  THEN  -
   0< IF  2DROP R> 2+  ELSE  R> @  THEN  >R ;
: [NEXT]   COMPILE (NEXT) , ;






```

```
     75
( Variables)
: INTEGER  <BUILDS 0 ,  IMMEDIATE  DOES>  [COMPILE] LITERAL
   ADDRESS @ IF  0 ADDRESS !  ELSE  COMPILE @  THEN ;

: (ARRAY) ( a a - a p )
   SWAP >R   7 DEFER  R> [COMPILE] LITERAL
   ADDRESS @ IF  0 ADDRESS !
   ELSE  ' @  7 #( @ +  2SWAP  THEN ;
: [+] ( i a - a)   SWAP 1- 2* + ;
: ARRAY ( n - )  <BUILDS  2* ALLOT IMMEDIATE
   DOES>  ' [+] (ARRAY) ;

: [*+] ( x y a - a)  ROT ROT >R  1-  OVER @ *  R> +  2* + ;
: 2ARRAY ( y x - )   <BUILDS  DUP , * 2* ALLOT IMMEDIATE
   DOES>  ' [*+] (ARRAY) ;
```

```
     76
( Statement numbers)
: FIND ( n - a)   1 #S @ #S 2+ DO
      OVER I @ = IF  2DROP I 2+ 0  LEAVE  THEN 4 +LOOP
   IF  0 SWAP  #S @ 2!  #S @ 2+  4 #S +!  THEN ;

: RESOLVE ( n)   FIND  DUP @  DUP 0< ABORT" duplicated"
   BEGIN  ?DUP  WHILE  DUP @  HERE ROT !  REPEAT
   HERE NEGATE  SWAP ! ;
: CHAIN ( n - a)   FIND  DUP @ 0< IF  @ NEGATE
   ELSE  DUP @  HERE ROT !  THEN ;

: STATEMENT ( n)   HERE 2- @ >R  -4 ALLOT  RPN CFA EXECUTE
   R> RESOLVE  [COMPILE] START ;



```

```
     77
( BASIC )
: LET   STATEMENT  1 ADDRESS ! ; IMMEDIATE
: FOR   [COMPILE] LET ;  IMMEDIATE
: [1]   COMPILE 1  HERE ;
: TO    RPN DROP  ' [1] 0 ;  IMMEDIATE
: STEP  RPN DROP  ' HERE 0 ;  IMMEDIATE
: NEXT  STATEMENT 2DROP ' [NEXT] 0   1 ADDRESS ! ; IMMEDIATE
: REM   STATEMENT  IN @ C/L / 1+ C/L * IN ! ; IMMEDIATE
: DIM   [COMPILE] REM ; IMMEDIATE
: STOP  STATEMENT  COMPILE ;S  ; IMMEDIATE
: END   STATEMENT 2DROP [COMPILE] ; [COMPILE] FORTH ; IMMEDIATE

: (GOTO) (GET#)   COMPILE JUMP  CHAIN , ;
: GOTO   STATEMENT  (GOTO) ;  IMMEDIATE
: IF     STATEMENT  LOGIC ;  IMMEDIATE
: THEN   RPN 0  COMPILE SKIP  (GOTO) ; IMMEDIATE
```

Michael Perry 1981

```
    78
( Charles Moore's BASIC compiler, Input and Output )
: ASK   ." ? " QUERY ;
: PUT   (GET#) SWAP ! ;
: (INPUT)   COMPILE PUT ;
: (,) ( n)   (.)  14 OVER - SPACES  TYPE SPACE ;
: , ( n)   ?IGNORE   ' (,) 1 DEFER ;  IMMEDIATE
: "   [COMPILE] ."  2DROP ;  IMMEDIATE
INPUT DEFINITIONS
: ,   ?IGNORE  RPN 0  (INPUT)  1 ADDRESS ! ;  IMMEDIATE

ARITHMETIC DEFINITIONS
: PRINT   STATEMENT  COMPILE CR  ' (,) 1 ;  IMMEDIATE
: INPUT   STATEMENT  2DROP  COMPILE ASK  ' (INPUT) 0  INPUT
   1 ADDRESS ! ;  IMMEDIATE
```

```
    79
( Operators)
LOGIC DEFINITIONS
: <> [ n n - t ]  = NOT ;   2 PRECEDENCE <>
: <= [ n n - t ]  > NOT ;   2 PRECEDENCE <=
: >= [ n n - t ]  < NOT ;   2 PRECEDENCE >=
                            2 PRECEDENCE =

ARITHMETIC DEFINITIONS
: = ( a n)   SWAP ! ;   1 PRECEDENCE =
: ** ( n n - n)   1  SWAP 1 DO  OVER * LOOP  * ;
6 PRECEDENCE ABS
5 PRECEDENCE **
4 PRECEDENCE *   4 PRECEDENCE /   4 PRECEDENCE */
3 PRECEDENCE +   3 PRECEDENCE -
2 PRECEDENCE <   2 PRECEDENCE >
```

```
    80
[ Dwyer, page 17, Program 1]    SCR
INTEGER J   INTEGER K

: RUN   START
10 PRINT " THIS IS A COMPUTER"
20 FOR K = 1 TO 4
30 PRINT " NOTHING CAN GO"
40 FOR J = 1 TO 3
50 PRINT " WRONG"
60 NEXT J
70 NEXT K
80 END

RUN
```

```
    81
[ basic: array demo ]   SCR
INTEGER K
9 ARRAY COORDINATE

: RUN   START
10 FOR K = 1 TO 9
20 LET COORDINATE K = ( 10 - K ) ** 3
40 PRINT COORDINATE K + 5
60 NEXT K
80 END

RUN
```

```
    82
[ basic: input/print ]   SCR
INTEGER K
INTEGER X
INTEGER Y

: RUN   START
10 INPUT X , Y
20 LET K = X * Y ** 3
40 PRINT X , Y , K
90 END
;S
```

!

```
**************************************************
*                                                *
*           Michael Perry                        *
*           1446 Stannage Ave.                   *
*           Berkeley, Calif. 94702               *
*           (415) 526-8696                       *
*                                                *
**************************************************
```

Michael Perry 1981

A ROUNDTABLE ON RECURSION

Recursion, as it applies to FORTH, is the technique of defining a word in such a way that it calls itself. One of the nicest examples I've seen of a good use for recursion can be found in Douglas R. Hofstadter's book Godel, Escher, Bach. He describes a system which can produce gramatically correct phrases out of parts of speech.

I'll use FORTH to describe his example:

```
: FANCY-NOUN
   4 CHOOSE
   (select random number 0-3)
   CASE
     0  OF  NOUN  ENDOF
     1  OF
        NOUN  PRONOUN
        VERB  FANCY-NOUN  ENDOF
     2  OF
        NOUN  PRONOUN
        FANCY-NOUN  VERB  ENDOF
     3  OF
        NOUN  PREPOSITION
        FANCY-NOUN  ENDOF
   ENDCASE ;
```

Three of the four possible variations on FANCY-NOUN include a call on FANCY-NOUN itself. Case 0 might produce "books." Case 1 might produce "man who reads books." But Case 1 might also produce something more complicated, like "man who reads books that explain algebra," if the iner call to FANCY-NOUN decides to get fancy.

Normally FORTH deliberately prevents recursion so that you can call an existing word inside the definition of a new definition of the same name. For example:

```
: +  SHOW-STACK  +  SHOW-STACK ;
```

This example might be a redefinition of plus to teach beginners what the stack looks like before and after addition. The plus that is called in the middle of the definition is the original + , not the one being defined.

FORTH prevents recursion with a word called SMUDGE . This word usually toggles a bit in the name field of the word most recently defined. With this bit toggled, the name is "smudged"; that is, unrecognizable. In the definition of + above, the colon lays down a head in the dictionary, and then executes SMUDGE before compiling the rest of the definition.

When the second + is encountered, the compiler searches the dictionary for a word of that name. The new head with the same name is bypassed only because it has been smudged.

At the end of the definition, semicolon again executes SMUDGE . This toggles the bit back to its original state, so that the name is again findable.

There are various means of circumventing FORTH's protection against recursion. Here are two recent contributions from our readers:

A Recursion Technique

Christoph P. Kukulies
Aachen, West Germany

Here is my solution to the problem of recursion in FORTH shown in a possible way to implement the ACKERMANN's function (see FORTH DIMENSIONS, Vol. III, No. 3, p. 89).

First test if your FORTH-system is "crash-proof" with the following sequence:

```
: CRASH  [ SMUDGE  ] CRASH ;
SMUDGE CRASH
```

After having recovered from CRASH you should try this:

```
(m n -> ACKERMANN (m,n)
:ACKERMANN  (  m n -- ACK)
[ SMUDGE ] SWAP DUP 0= IF  DROP 1+
   ELSE SWAP DUP
     0= IF DROP 1 - 1 ACKERMANN
        ELSE OVER SWAP
          1 - ACKERMANN SWAP
          1 - SWAP ACKERMANN
        THEN
   THEN ; SMUDGE
```

Be aware of typing
3 4 ACKERMANN .

Another Recursion

Arthur J. Smith
Osahawa Canada LIG 6P7

Regarding the recursion problem, I think that I have found a more elegant solution. The solution involves an immediately executed word to re-SMUDGE the word being defined.

I define a word RECURS as follows:

```
: RECURS   SMUDGE ;   IMMEDIATE
```

then use the word to bracket the recursive self definition as in the example:

```
: SUM
  DUP 1- DUP IF RECURS SUM RECURS
  ENDIF
  +
;
```

I use the RECURS word in tree searches.

Editor's note:

The technique that is generally preferred was described by Joel Petersen in the original article. It defines MYSELF as

```
:   MYSELF
      LATEST  PFA  CFA , ; IMMEDIATE
```

or, for some other versions such as polyFORTH:

```
:   MYSELF
      LAST  @  @  2+  ,  ;   IMMEDIATE
```

MYSELF simply compiles the code field of the latest header in the dictionary (the word being defined) into the definition.

The problem with using the word SMUDGE inside a definition is 1) it's not readable, since smudging has nothing to do with what the definition is about, and 2) its behavior is different on different systems.

Similarly, having to say RECURS ACKERMANN RECURS is not quite as readable as simply MYSELF.

An even more readable solution is this:

```
:  :R
   [COMPILE]  :  SMUDGE ;  IMMEDIATE
:  R;
   SMUDGE  [COMPILE] ; ;  IMMEDIATE
```

Here a special version of colon and of semi-colon named :R and R; are defined to allow recursion without any other hoopla.

8080 ASSEMBLER

John J. Cassady
339 15th Street
Oakland, CA 94612

This 8080 assembler has been available in a slightly different form for approximately one and one-half years. It appears to be bug-free.

ENDIF 's have been replaced by THEN, and AGAIN has been removed in conformance with FORTH-79. I have never had occasion to use AGAIN ; I doubt if I'll miss it.

I have removed the compiler security. We frequently want non-structured control mechanisms at the code level. The ?PAIRS really gets in the way.

I have introduced three macros: NEXT PSH1 and PSH2. They emplace, respectively, a jump to NEXT , a jump to the byte before NEXT and a jump to two bytes before NEXT . Literally, PSH1 means push one level (HL) and fall into NEXT . I believe this is a more traditional approach and the source code has a cleaner appearance.

The actual address of NEXT is stored in (NEXT) . Its value is plucked from ;S . This technique was suggested by Patrick Swayne of the Heath User's Group. I say "suggested" because Swayne's method is a bit different.

I have left out the conditional CALLs. I never used them and they can always be " C, " 'd in. The conditional jumps are, of course, handled automatically by the conditionals: IF WHILE and UNTIL, in conjunction with the flag testers: 0 = CS PE 0 < and NOT .

I have opted to retain the immediate instructions MVI and LVI as opposed to an immediate flag #.

The 1MI 2MI etc stands for "number one machine instruction" etc. The first cut of this assembler was written when three letter names were the craze.

I have a selfish motive in publishing this assembler. I hope that this will flush out assemblers for other processors and that there will be a "rush to publish." There is a good reason to do this besides vanity. If someone else publishes the assembler for the "xyz" chip that you use, and it becomes established, it means that you will have to change your code to conform with the quirks of the "established" version. It pays to get there first.

```
Screen   48   30H
( FIGFORTH 8080 ASSEMBLER 1                    81AUG17   JJC 80MAR04 )
HEX VOCABULARY ASSEMBLER IMMEDIATE : 8* DUP + DUP + DUP + ;
' ASSEMBLER CFA ' ;CODE 8 + !          ( PATCH ;CODE IN NUCLEUS )
: CODE ?EXEC CREATE [COMPILE] ASSEMBLER !CSP ; IMMEDIATE
: C; CURRENT @ CONTEXT ! ?EXEC ?CSP SMUDGE ; IMMEDIATE
: LABEL ?EXEC 0 VARIABLE SMUDGE -2 ALLOT [COMPILE] ASSEMBLER
   !CSP ; IMMEDIATE    ASSEMBLER DEFINITIONS
4 CONSTANT H    5 CONSTANT L    7 CONSTANT A    6 CONSTANT PSW
2 CONSTANT D    3 CONSTANT E    0 CONSTANT B    1 CONSTANT C
6 CONSTANT M    6 CONSTANT SP    ' ;S 0B + @ CONSTANT (NEXT)
: 1MI <BUILDS C, DOES> C@ C, ; : 2MI <BUILDS C, DOES> C@ + C, ;
: 3MI <BUILDS C, DOES> C@ SWAP 8* + C, ;
: 4MI <BUILDS C, DOES> C@ C, C, ;
: 5MI <BUILDS C, DOES> C@ C, , ;  : PSH1 C3 C, (NEXT) 1 - , ;
: PSH2 C3 C, (NEXT) 2 - , ;       : NEXT C3 C, (NEXT) , ;
;S

Screen   49   31H
( FIGFORTH 8080 ASSEMBLER 2                    81MAR22   JJC 80MAR04 )
00 1MI NOP      76 1MI HLT      F3 1MI DI       FB 1MI EI
07 1MI RLC      0F 1MI RRC      17 1MI RAL      1F 1MI RAR
E9 1MI PCHL     F9 1MI SPHL     E3 1MI XTHL     EB 1MI XCHG
27 1MI DAA      2F 1MI CMA      37 1MI STC      3F 1MI CMC
80 2MI ADD      88 2MI ADC      90 2MI SUB      98 2MI SBB
A0 2MI ANA      A8 2MI XRA      B0 2MI ORA      B8 2MI CMP
09 3MI DAD      C1 3MI POP      C5 3MI PUSH     02 3MI STAX
0A 3MI LDAX     04 3MI INR      05 3MI DCR      03 3MI INX
0B 3MI DCX      C7 3MI RST      D3 4MI OUT      DB 4MI IN
C6 4MI ADI      CE 4MI ACI      D6 4MI SUI      DE 4MI SBI
E6 4MI ANI      EE 4MI XRI      F6 4MI ORI      FE 4MI CPI
22 5MI SHLD     2A 5MI LHLD     32 5MI STA      3A 5MI LDA
CD 5MI CALL     ;S



Screen   50   32H
( FIGFORTH 8080 ASSEMBLER 3                    81AUG17   JJC 80MAR04 )
C9 1MI RET      C3 5MI JMP      C2 CONSTANT 0=  D2 CONSTANT CS
E2 CONSTANT PE  F2 CONSTANT 0<    : NOT 8 + ;
: MOV 8* 40 + + C, ;  : MVI 8* 6 + C, C, ;  : LXI 8* 1+ C, , ;
: THEN HERE SWAP ! ;                : IF C, HERE 0 , ;
: ELSE C3 IF SWAP THEN ;            : BEGIN HERE ;
: UNTIL C, , ;                      : WHILE IF ;
: REPEAT SWAP C3 C, , THEN ;
;S
```

```
Screen   51   33H
( EXAMPLES USING FORTH 8080 ASSEMBLER 1 81AUG17   JJC 80MAR12 )
FORTH DEFINITIONS HEX
CODE CSWAP ( WORD-1--- SWAPS HI AND LOW BYTE OF WORD ON STACK )
    H POP  L A MOV  H L MOV  A H MOV  PSH1  C;
CODE LCFOLD     ( FROM-2 QTY-1--- CONVERTS LOWER CASE TO UPPER )
    D POP  H POP
    BEGIN  D A MOV  E ORA  0=  NOT
    WHILE  M A MOV  60 CPI  CS  NOT
        IF  20 SUI  A M MOV
        THEN  D DCX  H INX
    REPEAT  NEXT  C;
;S




Screen   52   34H
( EXAMPLES USING FORTH 8080 ASSEMBLER 2 81AUG17   JJC 80MAR12 )
CODE CMOVE           ( FROM-3 TO-2 QTY-1--- SAME AS IN NUCLEUS )
    C L MOV  B H MOV  B POP  D POP  XTHL
    BEGIN  B A MOV  C ORA  0=  NOT
    WHILE  M A MOV  H INX  D STAX  D INX  B DCX
    REPEAT  B POP  NEXT  C;
CODE -CMOVE      ( FROM-3 TO-2 QTY-1--- SAME BUT OPP DIRECTION )
    C L MOV  B H MOV  B POP  XCHG
    H POP  B DAD  XCHG  XTHL  B DAD
    BEGIN  B A MOV  C ORA  0=  NOT
    WHILE  H DCX  M A MOV  D DCX  D STAX  B DCX
    REPEAT  B POP  NEXT  C;
: MOVE    ( FROM-3 TO-2 QTY-1--- SMART MOVE, DOES NOT OVERLAY )
    >R  2DUP  R>  ROT  ROT  -
    IF  -CMOVE  ELSE  CMOVE  THEN  ;
;S
Screen   53   35H
( EXAMPLES USING FORTH 8080 ASSEMBLER 3 81AUG17   JJC 80MAR12 )
80 CONSTANT CMMD  ( COMMAND BYTE )
F0 CONSTANT CMMDPORT  ( COMMAND PORT )
F1 CONSTANT STATUSPORT  ( STATUS PORT )
LABEL DELAY   ( --- DELAY CONSTANT IN DE, DON'T USE THE STACK )
    BEGIN  D DCX  D A MOV  E ORA  0=  UNTIL  RET  C;
CODE STATUS                                  ( BIT MASK-1--- )
    H POP  CMMD A MVI  CMMDPORT OUT
    1234 D LXI  DELAY CALL
    BEGIN
      STATUSPORT IN  L ANA  0=  NOT
    UNTIL  NEXT  C;
;S
```

Sieve of Eratostenes in FORTH

Mitchell E. Timin
Timin Engineering Co.

The enclosed version of Eratosthenes Sieve was written for an implementation of Timin FORTH release 3. I was pleased that it executed in 75.9 seconds, as compared to the 85 seconds of figFORTH. Mine was run on a 4 MHZ Z-80 machine, as were the others in the BYTE magazine article.

The speed improvement is primarily due to the array handling capability of Timin FORTH release 3. FLAGS is created with the defining word STRING; n FLAGS leaves the address of the nth element of FLAGS. This calculation occurs in machine code.

```
SCR # 35
( The Sieve of Eratosthenes, after J. Gilbreath, BYTE  9/81 )
8190 CONSTANT SIZE   SIZE STRING FLAGS   ( make array of flags )
: PRIME    O FLAGS   SIZE 1 FILL   ( start by setting the flags)
   0            ( create counter which remains on top of stack )
   SIZE  0  DO         ( repeat following loop 8190 times )
       I FLAGS   C@  ( fetch next flag to top of stack )
       IF             ( if flag is true then do the following: )
           I  DUP  +   3  +     ( calculate the prime number )
           DUP  I +             ( stack is: counter, prime, K )
           BEGIN  DUP  SIZE  <  WHILE   ( repeat for K < 8190 )
               0   OVER FLAGS  C!        ( clear Kth flag )
               OVER  +                   ( add prime to K )
           REPEAT
           DROP  DROP  1+  ( drop K & prime, increment counter)
       ENDIF
   LOOP   3 SPACES   .  ." PRIMES  " ; ( finish, display count)
SCR # 36
( testing the sieve algorithm )     O VARIABLE KOUNT
: BELL 7 EMIT ;
: NEW-LINE  CR  0 OUT !   ;
: NEW-LINE?   OUT @  70 >  IF  NEW-LINE  ENDIF   ;

: PRIME-TEST                   BELL   ( first sound the bell )
   10 0 DO  PRIME  LOOP   BELL   ( run the prime finder 10 X )
( above is for timing test, below is for validation )
   0  KOUNT  !   NEW-LINE       ( clear counter, start new line )
   SIZE  0  DO                  ( check each flag )
       I FLAGS  C@              ( see if it's set )
       IF  I DUP +  3 +         ( calculate the prime number )
           7 .R   NEW-LINE?     ( display it )
           1  KOUNT  +!         ( count it )
       ENDIF
   LOOP   CR  KOUNT ?   ." PRIMES   "   ;  ( display the count)
```

SKEWED SECTORS FOR CP/M

Roger D. Knapp

In regard to Michael Burton's article in FORTH DIMENSIONS, III/2, page 53, "Increasing fig-FORTH Disk Access Speed," I enclose a simple mod to the 8080 or Z80 assembly list to effect the CP/M skewed sector disk I/O. The FORTH routines I used to test the scheme are included. The first cluster or screen is offset by 52 sectors so that the operating system is transparent and screens 0 and 1 hold the directory. I move the message screens to SCR# 24 and 25 leaving 2-20 for the FORTH binary program run by CP/M or CDOS.

In order to check any increase in disk access speed I timed the following operation with a 10 screen buffer:

20 270 10 MCOPY 20 270 10 MCOPY
20 270 10 MCOPY

Elapsed times were 204 and 138 seconds for straight and skewed sectors respectively. Note that this reflects disk access speed for read/write of several sequential sectors and in no way compensates for inadequate planning or poor programming in other disk I/O applications.

If this seems trivial, then you have no need for CP/M file compatible I/O. My motive for these changes is the desire to write the assembler program for fig-FORTH via modem (easy to implement in FORTH) to friends and colleagues. As added value my disk I/O can be faster.

```
20T
        LD      DE,SETDSK       ; SEND DRIVE # TO CP/M
        CALL    IOS
        POP     BC              ; RESTORE (IP)
        JP      NEXT
;
TRTBL:  DB      0,1,7,13,19,25,5,11,17,23,3,9
        DB      15,21,2,8,14,20,26,6,12,18,24,4,10,16,22
;
        DB      86H     ; S-SKEW
        DB      'S-SKE'
        DB      'W'+80H                                       ADDED
        DW      SETDRV-12                                      AFTER
SSKEW:  DW      $+2                                         "SET DRIVE"
        POP     DE      ; SECTOR SEQUENTIAL
        LD      HL,TRTBL            ; TRANSLATION TABLE ABOVE
        ADD     HL,DE               ; ADDR OF NEW SECTOR
        LD      E,(HL)
        PUSH    DE                  ; SECTOR TRANSLATED
        JP      NEXT
;
*10T
;
        DB      87H             ; T&SCALC
        DB      'T&SCAL'
        DB      'C'+80H
        DW      SSKEW-9                  ; MODIFIED
TSCALC: DW      DOCOL,DENSTY
        DW      AT
        DW      ZBRAN,TSCALS-$
;       DOUBLE DENSITY
        DW      LIT,BUPDR2
*20T
;       SINGLE DENSITY
TSCALS: DW      LIT,52,PLUS              SKIP 52 SECTORS FOR
        DW      LIT,BUPDR1               OPERATING SYSTEM
        DW      SLMOD
        DW      LIT,MXDRV-1
        DW      MIN
        DW      DUP,DRIVE
        DW      AT,EQUAL
        DW      ZBRAN,TSCAL3-$
        DW      DROP
        DW      BRAN,TSCAL4-$
TSCAL3: DW      DRIVE,STORE
        DW      SETDRV
TSCAL4: DW      LIT,SEPTR1
        DW      SLMOD,TRACK
        DW      STORE,ONEP
        DW      SSKEW           ; SEQUENTIAL TO CP/M SKEW
        DW      SEC,STORE
        DW      SEMIS
;
*
```

```
SCR # 60
  ( CP/M style disk layout and I/O                      )
 FORTH DEFINITIONS DECIMAL

 LABEL IOS ( CP/M SERVICE REQUEST )  1 LDHLM, D  ADDP, JPHL, C;

 CODE SET-IO ( sector track addrs --- )
  H  POP, B  PUSH, H B LD, L C LD, 21 D LDPI, IOS CALL, B POP,
  H  POP, B  PUSH, H B LD, L C LD, 1B D LDPI, IOS CALL, B POP,
  H  POP, B  PUSH,         L C LD, 1E D LDPI, IOS CALL, B POP,
  NEXT, C;

 CODE SET-DRIVE  ( n --- )
   H  POP, B  PUSH,         L C LD, 18 D LDPI, IOS CALL, B POP,
  NEXT, C;


```

```
SCR # 61
  0 ( SECTOR SKEW FOR CP/M FORMAT CLUSTERS                              )
  1 FORTH DEFINITIONS DECIMAL
  2 : CTABLE ( bytesize TABLE )
  3     <BUILDS 0 DO C, LOOP DOES> + C@ ;
  4 22 16 10 4 24 18 12 6 26 20 14 8 2 21 15 9 3 23 17 11 5 25 19
  5 13 7 1 0 27 CTABLE S-SKEW  ( for CP/M clusters )
  6
  7 : NSETUP ( Setup n sectors for NXTS. )
  8 ( adrs blk n --- sec trk addr ... secn trkn addrn )
  9     ROT OVER 128 * + ROT ROT OVER + 1- SWAP 1- SWAP
 10     DO I  26 /MOD SWAP 1+ S-SKEW SWAP ROT 128 - DUP
 11     -1 +LOOP DROP ;
 12
 13 : NRTS ( Read n sectors. ) ( s t a ... sn tn an n --- )
 14     0 DO SET-IO SEC-READ DISK-ERROR @ IF LEAVE THEN LOOP ;
 15

SCR # 62
  0 ( MORE CP/M FORMAT DISK I/O                                         )
  1 FORTH DEFINITIONS DECIMAL
  2
  3 : NWTS ( Write n sectors to CP/M cluster. )
  4     0 DO SET-IO SEC-WRITE DISK-ERROR @ IF LEAVE THEN LOOP ;
  5
  6 : R/W-CP/M ( CP/M skewed cluster I/O.)
  7 ( addrs blk f --- )     >R  52 + 2000 /MOD  SET-DRIVE
  8         SEC/BLK NSETUP ( 52 + so cluster alloc CP/M    )
  9         R>  IF      SEC/BLK NRTS
 10             ELSE    SEC/BLK NWTS
 11             ENDIF  DISK-ERROR @ 8 ?ERROR ;
 12
 13 ( All of screens 61 and 62 shamelessly adapted from John James')
 14 ( fig-FORTH for the LSI-11.                                         )
 15
ok
```

Diagnostics on Disk Buffers

Timothy Huang
9529 NE Gertz Circle
Portland, OR 97211

While I was in the process of explaining the disking to some friends, I found it would be nice to show them some sort of representation which lists all the disk buffer status. This short program was then written for this purpose.

The figFORTH uses the memory above USER area for the disk buffer. This disk buffer area is further divided into several blocks with the length of each block equal to B/BUF + 4 bytes. There are some implementations that set B/BUF to be 1024 bytes and some, like 8080 CP/M, that set it to be 128 bytes. Another constant beside B/BUF frequently referred in disking is the B/SCR (buffers per screen). For B/BUF = 1024, the B/SCR = 1 and for B/BUF = 128, B/SCR = 8.

Each block needs 2 bytes in front of it as the header which contains the update bit (bit 15) and block number (lower 0-14 bits). It also needs a 2-byte tail to end the block.

The word BLOCK will put the beginning address of a given block (assuming the block number on stack before executing BLOCK). With these simple words, virtual memory can be utilized, but it is beyond the scope of this short article.

The short program will display the status of each disk block until it is exhausted or you terminate it by pressing any key. The first thing it does is print out the title line (line 4). Line 5 sets up the boundary for the DO ... LOOP. Line 6 prints the buffer number while line 7 prints the beginning address of each buffer in hex. Lines 8 and 9 check the buffer update status. If it has been updated, then an " : " will be printed in the upd column. Lines 10 and 11 calculate the block number, screen number and the -sub number. The reason for teh -sub is because for my system, B/LBUF = 128, B/SCR = 8, there are 8 blocks to make a whole screen. So, I thought it would be handier to know which subpart of a given screen the block I want.

Lines 12 and 13 check the early termination and finish the definition.

```
SCR # 90

  0 ( .BUFS         TDH        7/11/81     )
  1 DECIMAL
  2 : .BUFS ( display adr of all buffers  )
  3     CR ." #  Addr(hex) Upd  Block#  Screen    -sub"
  4     FIRST  #BUFF 1+ 1 DO
  5     CR I 2 .R  2 SPACES
  6     DUP 2+  HEX 6 0 SWAP D.R  DECIMAL  3 SPACES
  7     DUP @  32768 AND
  8        0= 0= 32 + EMIT  2 SPACES
  9     DUP @ 32767 AND DUP  6 .R  4 SPACES
 10        B/SCR  /MOD 5 .R  4 SPACES  2 .R
 11     132 +  ?TERMINAL IF LEAVE THEN
 12     LOOP DROP CR ;
 13
 14
 15


OK
.BUFS
#  Addr(hex) Upd  Block#  Screen   -sub
 1     3E82          720       90    0
 2     3F06          721       90    1
 3     3F8A          722       90    2
 4     400E          723       90    3
 5     4092          724       90    4
 6     4116          725       90    5
 7     419A          726       90    6
 8     421E          727       90    7
 9     42A2            0        0    0
```

FLOATING POINT ON THE TRS-80

Kalman Fejes
Kalth Microsystems
PO Box 5457, Station F
Ottawa, Ontario K2C 3J1
Canada

Most FORTH systems have no provisions for handling floating piont numbers, although most popular micros have the necessary routines hidden in their ROM-based BASIC interpreter. These are fast routines written in assembler. The following is to demonstrate how these can be accessed and used to implement single precision floating pint arithmetics for the TRS-80 in MMSFORTH, Version 1.8.

Single precision floating point data is stored as a normalized binary fraction, with an assumed decimal point before the most significant bit. The most significant bit also doubles as a sign bit.

A binary exponent takes one byte in each floating point number. It is kept in excess 128 form; that is, 128 is added to the actual binary exponent needed.

The binary mantissa is 24 bits long, the most significant bit representing the sign bit. It is stored as 3 bytes normally with the least significant byte (LSB) stored first and the most significant byte (MSB) last, followed by the exponent.

Numbers should be entered using the notation specified for the TRS-80 L2 BASIC. Integers and dobule precision numbers are converted to and stored internally as single precision numbers.

The complete vocabulary and listing of the source screens for either MMSFORTH or figFORTH (specify) is available for $7 (U.S.) from Kalth microsystems. It includes both single and double precision, trigonometric and log functions, floating point constant, variable and stack operators, conversion routines to/from integers (FORTH type) and floating piont numbers.

BLOCK 9

```
( FTP #1    :KIF 810816)    FORGET FTASK : FTASK ;    HEX
( SINGLE PREC. FLOATING POINT FOR TRS-80 IN MMSFORTH V1.8)
: EXX   D9 C, ;
CODE F.&     EXX  OFBD CALL    28A7 CALL    EXX    NEXT
CODE F#&     EXX  HL POP    2 RST     0E6C CALL
                  0AB1 CALL    EXX    NEXT
: F@    DUP 2 + @  SWAP @  4 40AF C! ;
: F!    DUP ROT SWAP !  2 + !  4 40AF C! ;
: A S    4121 F@ ;
: F#0    HERE  0 OVER 3E  FILL  BL WORD  F#&  A S ;
: F#IN    "  ? "  PAD  DUP 1+  63  EXPECT  F#&  A S ;
: F#1    F#0  SWAP (L) (L) , ,   (L) (L) , , ;
: F#    STATE C@  IF  F#1  ELSE  F#0  THEN ;   IMMEDIATE
: F.    S A   F.&  4 40AF C! ;
: 10FT ;                                      DECIMAL

```

BLOCK 10

```
( FLOT. PT. #2  :IF 810816)  FORGET 10FT   : 10FT ;
HEX
CODE F+&    EXX    DE POP    BC POP    716 CALL    EXX    NEXT
CODE F-&    EXX    DE POP    BC POP    713 CALL    EXX    NEXT
CODE F*&    EXX    DE POP    BC POP    847 CALL    EXX    NEXT
CODE F/&    EXX    DE POP    BC POP    8A2  CALL    EXX   NEXT
: F+    S A  F+&  A S ;      : F-   S A  F-&  A S ;
: F*    S A  F*&  A S ;      : F/   S A  F/&  A S ;
DECIMAL
           ( SAMPLE AND TEST ROUTINES)
: FTEST    F#IN  CR    F#  2  F+  F# 200.0E-2  F-
           F# 5000.1  F*  F# 5.0001E+3  F/
            PAD F!  PAD F@  F. ;
;S


```

GLOSSARY

Single Precision Floating Point

F +	(F1 F1 -- F) (F=F2+F1)	Add
F -	(F2 F1 -- F) (F=F1-F1)	Subtract
F *	(F2 F1 -- F) (F=F2*F1)	Multiply
F /	(F1 F1 -- F) (F=F2/F1)	Divide

F # (-- F)
Takes a number from the current buffer, converts it to single precision floating point number and leaves it on the stack.

F # IN (-- F)
Asks for a floating pint number from the keyboard, and leaves it on the stack.

F @ (A -- F)
Floating point fetch. Takes a floating point number from memory at address and leaves it on the stack.

F ! (F A --)
Floating point store. Stores the floating point number on stack in memory at location A.

F TEST (--)
A sample program to demonstrate the use of these floating point operators. It asks for a floating point number from the keyboard, manipulates it using all the operators defined and prints the result. (It should be the same number that was supplied.)

Notes: A -- 16 bit address

F, F1, F2 -- are single precision floating pint numbers (two 16-bit words each).

TURNING THE STACK INTO LOCAL VARIABLES

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

Occasionally in writing a definition, I find that I need to do unwieldly stack juggling. For example, suppose you come into a word with the length, width, and height of a box and want to return the volume, surface area, and length of edges. Try it!

For this kind of siuation I developed my ARGUMENTS-RESULTS words. The middle block fo the triad shows my solution to the box problem.

The phrase "3 ARGUMENTS" assigns the names of local variables 1 through 9 to nine stack positions, wtih S1, S2 and S3 returning the top 3 stack values that were there before 3 ARGUMENTS was executed. S4 through S9 are zero-filled and the stackpointer is set to just below S9.

S1 thorugh S9 act as local variables returning their contents, not their addresses. To write to them you precede them with the word " TO ". For example, 5 TO S4 writes a 5 into S4. Execution of S4 returns a 5 to the stack.

After all calculating is done, the phrase "3 RESULTS" leaves that many results on the stack relative to the stack position when ARGUMENTS was executed. All intermediate stack values are lost, which is good because you can leave the stack "dirty" and it doesn't matter.

```
SCR # B
              ( ***( ARGUMENTS-RESULTS )*** )
VARIABLE [ARG]       VARIABLE [TO]
: +ARG CREATE , DOES> @ [ARG] @ SWAP - [TO] @ ?DUP
       IF 0< IF +! ELSE ! ENDIF ELSE @ ENDIF 0 [TO] ! ;

 0 +ARG S1          2 +ARG S2         4 +ARG S3           6 +ARG S4
 8 +ARG S5          A +ARG S6         C +ARG S7           E +ARG S8
10 +ARG S9                ( *TO VARIABLES* )

:  TO  1 [TO] ! ;         ( *SETS  STORE FLAG FOR +ARG* )
: +TO -1 [TO] ! ;         ( *SETS +STORE FLAG FOR +ARG* )

: ARGUMENTS R> [ARG] @ >R >R 2* SP@ + DUP [ARG] ! 12 - SP@ SWAP
            - 2/ 0 DO 0 LOOP 0 [TO] ! ;
: RESULTS 2* [ARG] @ SWAP - SP@ - 2/
          0 DO DROP LOOP R> R> [ARG] ! >R ;
```

```
SCR # C
( ARGUMENT EXAMPLE --- BOX COMES IN WITH HEIGHT, LENGTH
& WIDTH AND LEAVES VOLUME, SURFACE AREA & LENGTH OF EDGES )

: BOX        3 ARGUMENTS
  ( VOLM )     S1 S2 S3 * * TO S4
  ( SURF )     S1 S2 2 * * S2 S3 2 * * S1 S3 2 * * + + TO S5
  ( EDGE )     S1 4 * S2 4 * S3 4 * + + TO S3
               S5 TO S2
               S4 TO S1
             3 RESULTS ;
```

```
SCR # 20
: TASK ;
: DISK@! 5 ARGUMENTS
             S1 S2 0400 U/MOD 1+ TO S1 TO S2
                BEGIN S4 0>
                   WHILE S1 BLOCK S2 + S3
                   S5 IF SWAP UPDATE ENDIF
                   S4 0400 S2 - MIN DUP TO S6 CMOVE
                   S6 +TO S3
                   S6 NEGATE +TO S4
                   1 +TO S1
                   0 TO S2
                REPEAT
          0 RESULTS ;
: DISK@ 0 DISK@! ;
: DISK! 1 DISK@! ;
-->
```

GRAPHIC GRAPHICS

Bob Gotsch
California College of Arts and Crafts

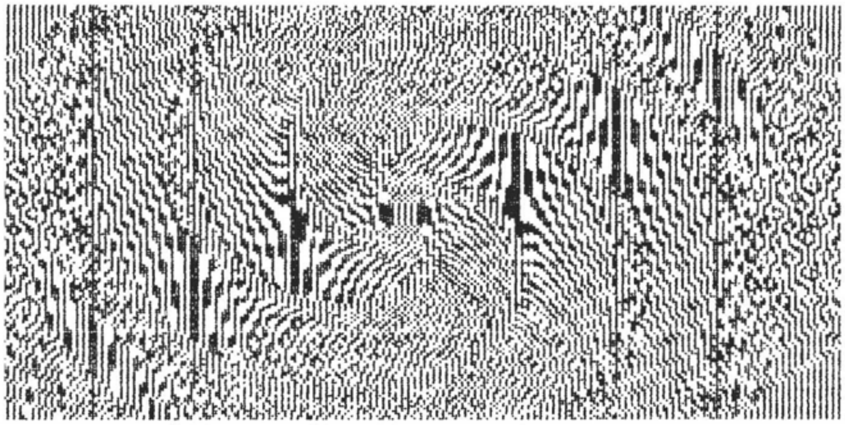
Accompanying these comments are several graphic specimens drawn on Apple computer using FORTH and printed on a dot-matrix printer. They range from logotype design to experiments in geometry and pattern. One can generate real-time motion graphics on the Apple in which color and action partially compensate for the low resolution of 280 by 192 pixels. Hardcopy, whether prinout or color photo, isn't the final product. The interactive, sequenced and timed display on the screen is the designed product, likely to displace the medium of print on paper in the future.

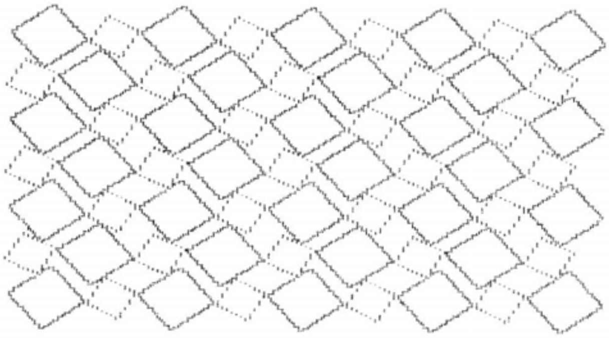
While these graphic samples could have been programmed in other languages, I have found the advantages of using FORTH are both practical and expressive: immediate and modular experimentation with the peculiarities and limitations of the Apple video display, and orchestration of complex visual effects with self-named procedures rather than the tedious plots and pokes to undistinguished addresses. With this ease of wielding visual ideas, FORTH might lead to a new era of computer graphics, even creative expression.

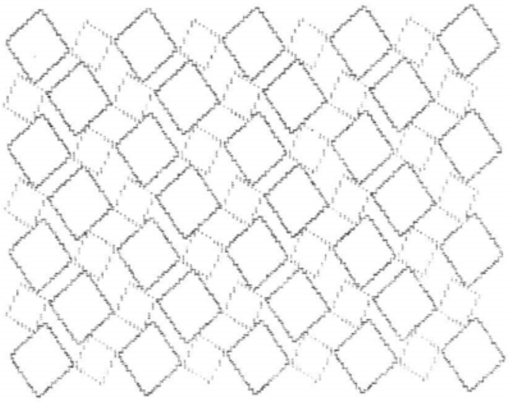
It may remain individual and personal expression, however, without graphics standards. Transportability of grahics--generating code may be neither possible nor desirable considering the differences in video display generation, alternate character sets, shape tables, display lists, interrupts, available colors, etc., between microcomputers. Each has some individual features to exploit. Most have, however, such limited memory for graphics as to make machine-dependent economy an overriding aspect of programming for graphics.

Despite the rarity of FORTH graphics thus far, I'm convinced it is an excellent vehicle for bringing out undiscovered graphics potential of each micro. In addition, the visibility gained by some effort to evolve grahic ideas in FORTH would help in both spreading and teaching the language. Perhaps this issue of FORTH DIMENSIONS will stimulate just such activity.

Editor's Note: The author tells me that Osborne/McGraw-Hill publishers have used his patterns, generated on Apple II using Cap'n Software FORTH, as cover artwork for their book "Some Common BASIC Programs".







CASES CONTINUED

Editor's Note: In Volume II, Number 3, FORTH DIMENSIONS published the results of FIG's CASE Statement Contest. As we had hoped, the variety of responses has stimulated further work on the subject. Here are four additional CASE constructes submitted by our readers.

Eaker's CASE for 8080

John J. Cassady

Here is an 8080 (Z80) version of the keyed case statement by Charles Eaker that was published in FORTH DIMENSIONS II/3, page 37. I have found it very useful.

```
( CASE STATEMENT BY CHARLES EAKER FD II 3 39       JJC 81AUG09 )
: CASE ?COMP CSP @ !CSP 4 ; IMMEDIATE
CODE (OF) H POP D POP ' - 8 + CALL L A MOV H ORA 0=
   IF B INX B INX NEXT ENDIF D PUSH ' BRANCH JMP C;
: OF 4 ?PAIRS COMPILE (OF) HERE 0 , 5 ; IMMEDIATE
: ENDOF 5 ?PAIRS COMPILE BRANCH HERE 0 ,
   SWAP 2 [COMPILE] THEN 4 ; IMMEDIATE
: ENDCASE 4 ?PAIRS COMPILE DROP
   BEGIN SP@ CSP @ = 0=
   WHILE 2 [COMPILE] THEN
   REPEAT CSP ! ; IMMEDIATE
: TEST CASE       41 OF ." A " ENDOF
                  42 OF ." B " ENDOF
                  65 OF ." e " ENDOF        ENDCASE ;
(  41 TEST A OK  )

```

Eaker's CASE Augmented

Alfred J. Monroe
3769 Grandview Blvd.
Los Angeles, CA 90066

I was delighted with Dr. Eaker's CASE construction (FORTH DIMENSIONS, Vol. II, No. 3, p. 37) and implemented it immediately. Recently I have found it desirable to augment CASE with three additional constructs in order to treat ranges of variables. It has occurred to me that other FORTH users may be interested in the same extension, hence this short note.

Screen 144 lists Dr. Eaker's CASE construct with one slight modification. OF has been modified to use (OF). The original OF compiled to ten bytes. The revised OF compiles to six bytes. This forty percent reduction in code is not as impressive as that which occurs using Dr. Eaker's CODE word (OF) construct, but it does have the advantage that it is highly portable. (OF) tests for equality and leaves a true or false flag on the stack. Note that it drops the test value if the test is true.

Screen 145 lists the extensions that I have found useful, <OF, >OF, and RNG-OF. <OF does a "less than" test. >OF does a "greater than" test. RNG-OF does an inclusive range test. <OF and >OF are trivial modifications of OF and (OF). RANGE and RNG-OF are constructed in the same spirit as (OF) and OF .

Screen 144 compiles to 175 bytes. Screen 145 compiles to 223 bytes.

```
SCR # 144
( DR. EAKER'S CASE CONSTRUCT WITH A SLIGHT MODIFICATION )
: CASE ?COMP CSP @ !CSP 4 ; IMMEDIATE
: (OF) OVER = IF DROP 1 ELSE 0 ENDIF ;
: OF 4 ?PAIRS COMPILE (OF) COMPILE 0BRANCH
   HERE 0 , 5 ; IMMEDIATE
: ENDOF 5 ?PAIRS COMPILE BRANCH HERE 0 , SWAP 2
         [COMPILE] ENDIF 4 ; IMMEDIATE
: ENDCASE 4 ?PAIRS COMPILE DROP BEGIN SP@ CSP @ = 0 =
           WHILE 2 [COMPILE] ENDIF REPEAT CSP ! ; IMMEDIATE







-->
```

```
SCR # 145
( THE <OF, >OF, AND RNG-OF EXTENSIONS )
: (<OF) OVER > IF DROP 1 ELSE 0 ENDIF ;
: <OF 4 ?PAIRS COMPILE (<OF) COMPILE 0BRANCH
    HERE 0 , 5 ; IMMEDIATE
: (>OF) OVER < IF DROP 1 ELSE 0 ENDIF ;
: >OF 4 ?PAIRS COMPILE (>OF) COMPILE 0BRANCH
    HERE 0 , 5 ; IMMEDIATE
: RANGE >R OVER DUP R> 1+ < IF SWAP 1- > IF DROP 1 ELSE 0
          ENDIF ELSE DROP DROP 0 ENDIF ;
: RNG-OF 4 ?PAIRS COMPILE RANGE COMPILE 0BRANCH HERE 0 , 5 ;
IMMEDIATE




-->
```

```
SCR # 146
< EXAMPLE USE OF AUGMENTED CASE >
48 CONSTANT "0"  57 CONSTANT "9"  65 CONSTANT "A"
70 CONSTANT "F"  13 CONSTANT "CR"
3  CONSTANT CNTRL-C

0 VARIABLE FLAG

: SYN-ERR CR ." SYNTAX ERROR, REENTER NUMBER " CR
          DROP DROP 0 "0" ;
: C-ABORT CR ." COMMAND ABORT " CR DROP DROP QUIT ;


: ?ABORT CNTRL-C = IF DROP CR ." COMMAND ABORT " CR QUIT
                   ELSE DUP ENDIF ;

-->
```

```
SCR # 147
< GET-HEX  LEAVE A HEX # ON TOP OF STACK >
< A PRE DR. EAKER SOLUTION TO AN INTERACTIVE TERMINAL INPUT >


: GET-HEX 0 FLAG ! 0 BEGIN KEY DUP DUP EMIT ?ABORT
          13 = IF 1 FLAG ! DROP
             ELSE DUP "0" < IF SYN-ERR
               ELSE DUP "9" > IF DUP "A" < IF SYN-ERR
             ELSE DUP "F" > IF SYN-ERR ENDIF
             ENDIF ENDIF ENDIF ENDIF
 FLAG @ 0= IF 48 - DUP 9 > IF 7 - ENDIF SWAP 16 * + ENDIF

   FLAG @ UNTIL ;
```

```
SCR # 148
< A NEATER SOLUTION TO THE TERMINAL INPUT ROUTINE >
: GET-HEX 0 FLAG !
    0 BEGIN KEY DUP DUP EMIT
           CASE CNTRL-C OF C-ABORT        ENDOF
                  "CR"   OF 1 FLAG ! DROP ENDOF
                  "0"  <OF SYN-ERR        ENDOF
                  "F"  >OF SYN-ERR        ENDOF
             "9" 1 +   <OF 48 -           ENDOF
             "A" 1 -          >OF 55 -    ENDOF
                       SYN-ERR
          ENDCASE
FLAG @ 0= IF SWAP 16 * + ENDIF
FLAG @ UNTIL ;
```

```
SCR # 149
< A STILL NEATER SOLUTION >
: GET-HEX 0 FLAG !
 0 BEGIN KEY DUP DUP EMIT
 CASE CNTRL-C OF C-ABORT          ENDOF
       "CR"   OF 1 FLAG ! DROP    ENDOF
      "0" "9" RNG-OF 48 -         ENDOF
      "A" "F" RNG-OF 55 -         ENDOF
        SYN-ERR
 ENDCASE
FLAG @ 0= IF SWAP 16 * + ENDIF
FLAG @ UNTIL ;
```

Screen 147 illustrates a pre-Eaker solution to the design of an interactive terminal input that places a hexadecimal number on the stack, and which provides for error detection and error recovery. It is, of course written in my usual sloppy, unannotated, semi-readable fashion.

Screen 148 offers a neater solution in terms of <OF and >OF. It is definitely more readable. Screen 149 offers a still neater solution in terms of RNG-OF.

Screen 147 compiles to 160 bytes, screen 148 to 176 bytes, and screen 149 to 144 bytes. Need I say more?

CASE as a Defining Word

Dan Lerner

After reading the CASE contest articles and looking for a simple function, I am compelled to submit a simple CASE statement. These words are fast to compile and execute, compact, simple, generate minimum code, and very simple. There is no error checking since the form is so simple the most novice programmer can use it.

CASE is analogous to vectored GOTO in other languages. Its usage with my words is:

```
CASE          NAME
  A   IS      FUNCTION A
  B   IS      FUNCTION B
  C   IS      FUNCTION C
  (etc.)
  OTHERS ERROR FUNCTION
```

General usage would be as a menu selector; for example, you print a menu:

```
1   BREAKFAST
2   LUNCH
3   DINNER
```

SELECTION -->

The user types a number which goes n the stack, then executes the CASE word MEAL. MEAL selects BREAKFAST, LUNCH or DINNER, or ABORTS on error. The source is:

```
CASE          MEAL
  1   IS      BREAKFAST
  2   IS      LUNCH
  3   IS      DINNER
  OTHERS      NO MEAL
```

You have previously defined BREAKFAST, LUNCH, DINNER and NO MEAL.

```
106
( CASE        NAME
    A   IS      FUNCTION-A                     PAIR = VALUE-A
    B   IS      FUNCTION-B                            ADDR OF FUNCTION-A
    C   IS      FUNCTION-D
      ETC.
    OTHERS      ERRORFUNCTION    )

: CASE  CREATE  HERE 0. ,     ( AT COMPILATION BUILDS HEADER,LINK
                                   POINTS TO ADDR OF # OF PAIRS
                                   HERE SET TO ADDR OF VALUE-1 )
        DOES>                   ( AT EXECUTION, ADDR OF #OF PAIRS)
    1 ROT ROT DUP 2+ SWAP @
    0 DO 2DUP @ = IF DUP 2+ @    ( COMPAIRS INPUT VALUE          )
    EXECUTE ROT DROP 0 ROT ROT   ( WITH VALUE A, B, C, ETC, AND )
    LEAVE ELSE  2+ 2+ THEN LOOP ( EXECUTES ASSOCIATED FUNCTION )
    ROT IF @ EXECUTE ELSE DROP THEN DROP ;
```

```
107
(    CASE WORDS   )
: IS   , ' , 1+  ;    ( HERE, PAIR# -- HERE, NEXT-PAIR# )
: OTHERS  ' ,  SWAP ! ;        ( HERE, #-OF-PAIRS )
```

How CASE is Structured

CASE builds an array using IS and OTHERS to fill and complete the values in the array. At execution, the DOES> portion of CASE takes a value from the stack and looks through the array for it. A match executes the word, no match executes the word after OTHERS in source.

The form of CASE is a new class of words, as CONSTANT , VARIABLE , MSG , etc. are. The code executed to test the array is minimal.

Generalized CASE Structure in FORTH

E.H. Fey

Introduction

The CASE CONTEST held by FIG last year ended with some excellent contributions to the FORTH literature. The judges noted however that few people tried to devise a general case structure encompassing both the positional type, where the case is selected by an integer denoting its position in the list of cases (ala FORTRAN's computed GO TO), and the more general keyed type of structure, where the case selector key is tested for a match in the case words key list.

This article discusses a general case structure which combines the positional and keyed types. Like FORTH itself, the case structure is extensible. I have added a third type called range where the case selector key is tested to be within the range of pairs of values in the case words key list.

For any of the three types of structures, the user is also provided with the option of using headerless high level code sequences to specify the execution behavior of the individual cases.

A complete source listing in fig-FORTH is given on screens 165 to 180 with illustrative examples on screens 180 and 181. The source code listings may seem lengthier than usual but it is the author's practice to include the Glossary definition right with the source and to annotate the source code with notes on the status of the parameter stack. When this practice is followed, I find FORTH to be an emminently readable language, even months after the particular coding has been prepared. However, this style of coding requires a good FORTH video editor. With a good case structure in FORTH, that is not difficult to develop.

Background

In the Aug. '80 issue of Byte, Kim Harris introduced a very simple positional type of case compiler. A slightly revised version of his compiler is

```
: CASE: <LIST DOES> IX @ EXECUTE ;
```

where

```
: <LIST  <BUILDS SMUDGE !CSP ] ;
: IX ( k pfa...adr )  SWAP 1 MAX
  1 - DUP + + ;
```

and is used in the form:

```
CASE: xxxx cfa1 cfa2 .... cfan ;
```

to define a case selector word named xxxx.

When the new word, xxx , is executed in the form

```
k xxxx     ( k=1,2,...,n)
```

the k'th word in the list will be executed. For example, define the following words, COW , CHICK , PIG , and BARN :

```
    : COW     ." MooOOoo" ;
    : CHICK    ." Peep" ;
    : PIG     ." Oink" ;
 CASE: BARN    COW PIG CHICK ;
```

If we now execute the sequence 2 BARN , Oink will be typed. Similarly 1 BARN will type MooOOoo.

Although there are no error checks, this case structure is easy to use, executes fast and requires a minimum of dictionary space for each case word, xxx. Bilobran, etal have used CASE: extensively in developing a FORTH file system with named record components (1980 FORML proc. pp 188, Nov. 1980). I have done likewise following their example.

The interesting part of the definition of CASE: is the <BUILDS part which I have called <LIST for obvious reasons. It creates the dictionary entry for xxxx. Then, after executing SMUDGE and !CSP which are part of fig-FORTH's compiler security, it executes] which forces FORTH into the compilation state so that the user can enter the list. The list is terminated by ; which completes the definition of xxxx .

For CASE: words, the list is a list of code field addresses of previously defined FORTH words. Since FORTH is in the compilation state when the list is being entered, all the user has to do is list the names of the case select words (COW PIG CHICK in the example of BARN). FORTH then compiles their code field addresses, as long as they are not special IMMEDIATE words which execute during compilation.

Now suppose that we knew beforehand that the code field address of PIG was say 14382. The same definition of BARN could then have been achieved by

```
CASE: BARN  COW [ 14382 , ] CHICK ;
```

where [stopped the compilation state, 14382 was entered to the stack, the word , (comma) , compiled it and] resumed the compilation state.

The point is that <LIST is a powerful word for entering named lists and data of all sorts to the dictionary. The method of retrieval of the data is determined by the DOES> part of the compiler. Hence if we simply change the definition of the DOES> part of CASE: , we can transform it into a general purpose case compiler.

The Multi-Purpose Case Compiler

The method utilized to develop a generalized case compiler is to compile a number for the case type as the first byte in the parameter field of xxx . At execution time, the number is retrieved and used to select the appropriate DOES> part for the case type of xxxx . The type number is transparent to the user.

The definition of the new case compiler is:

```
: MCASE: <BUILDS SMUDGE !CSP
  HERE 1 C, 0 C, ]
      DOES> DUP C@ DOESPART ;
```

where DOESPART is a case selector word defined by CASE: .

The <BUILDS part of MCASE: compiles a "1" for the default case type (positional) and a "0" for the count of the number of cases entered into the case list. It also leaves the parameter field address of the newly defined word on the stack so that it can be found later during the compilation process even though its name field is smudged.

If the newly defined case word, say xxxx , is to be other than the positional type, it is immediately followed by the word KEYED or RANGE to define the type of xxxx as keyed type = 3 or range type = 5.

```
: KEYED   3 OVER C !;   IMMEDIATE
: RANGE   5 OVER C !;   IMMEDIATE
```

The case list subsequently entered must agree with the case type specified.

Two options are provided for the execution elements of the case list. The first or default option is the single word execution as in CASE: . The second option allows a headerless sequence of FORTH words to be defined as the execution elements of each case. The two may not be mixed.

A default case at the end of the case list is mandatory, although it may be a null word. The default case must be preceded by the word DEFAULT: whose definition is

```
: DEFAULT:  ?COMP EOL , HERE
  OVER C@ [DEF] ;  IMMEDIATE
```

where EOL is an end of list terminator constant defined by

```
' ;S CFA CONSTANT EOL
```

and [DEF] is a case selector word defined by CASE: .

DEFAULT: first checks to see that you are in the compile state since you should be compiling xxxx . It then enters the end of list terminator, EOL , to the dictionary. Finally it takes the parameter field address of xxxx left on the stack by the <BUILDS part of MCASE: , gets the type of xxxx and executes the case selector word [DEF] depending on the type of xxxx . If the type is 1, 3 or 5, [DEF] counts the number of cases entered and stores it in the second byte of the parameter field of xxxx . If the case type is 2, 4 or 6, then the execution elements are headerless code sequences. Hence for these types, [DEF] initiates the process of defining the default code sequence.

Execution of Case Selector

All case selector words, xxxx , defined by MCASE: are executed in the form:

```
k xxxx
```

where the key, k , is an integer. The interpretation of k in selecting the case depends on the case list type.

With three case list types and two options for each type, there are actually 6 different forms of case lists available. Let's consider first the lists with single word execution elements.

Single Word Execution Elements

(1) Positional type

MCASE: is used in the form:

```
MCASE: xxxx  cfa1 cfa2 ... cfan
  DEFAULT: cfad ;
```

When xxxx is executed in the form k xxxx , the case cfak will be selected if k=1, 2,...,n . Otherwise the default case, cfad, will be selected and executed.

(2) Keyed type

```
MCASE: xxxx KEYED
      [ k1  ,  ] cfa1
      [ k2  ,  ] cfa2
      ...
      [ kn  ,  ] cfan
      DEFAULT: cfad ;
```

When xxxx is later executed in the form k xxxx , the case cfai will be executed if a value of k=ki is found in the list. Otherwise, the default case, cfad , will be executed.

```
 ( GENERAL CASE STRUCTURE             EHF 10/23/81              )

( EXECUTION VARIABLES AND ARRAYS ala Kim Harris, Byte Aug '80 )
 ( pp 184 also see M. A. McCourt, FD II/4 pp 109.   EHF 2/11/81 )


: IX ( k pfa...adr ) ( Computes adr of index k = 1,2,...,n )
     SWAP 1 MAX                          ( ...pfa kmax1 )
     1 - DUP + + ;                       ( ...pfa+2[k-1] )

: <LIST ( General <BUILDS word to construct named lists )
   <BUILDS SMUDGE !CSP ] ;

 '  :  CFA @ CONSTANT COLON   ( For headerless code definitions)
  ' ;S  CFA   CONSTANT EOL     ( End of list delimiter )
                                                              -->
 : CASE: <LIST DOES> IX @ EXECUTE ;
  ( Used in the form:   CASE: xxxx cfa1 cfa2...cfan ;           )
  ( to create an execution array xxxx with initial values cfa1,)
  ( cfa2,..,cfan  which are code field addresses of previously )
  ( defined words. Executing xxxx in the form:   k xxxx         )
  ( will produce the execution of cfak , k= 1,2,...,n           )

 : LIST: <LIST DOES> IX @ ;
  ( Used in form:     LIST: xxxx [ n1 , n2 , n3 ,.... , ] ;     )
  ( to create a list of constants named xxxx . Executing xxxx  )
  ( in the form:   k xxxx will leave nK on the stack.          )

: XEQVAR: <LIST DOES> @ EXECUTE ;
  ( Used in the form:   XEQVAR:  xxxx  cfa ;                    )
  ( to create an execution variable xxxx with an initial value )
  ( cfa which is an existing word. Executing xxxx causes ) -->
  ( cfa to be executed. The word cfa may be changed by using   )
  ( INSTALL nnnn AT xxxxx  where nnnn is the new word.  )

: INSTALL ( ...cfa) [COMPILE] ' STATE @ IF COMPILE CFA ELSE CFA
  THEN ;       IMMEDIATE

: AT ( cfa...) [COMPILE] ' STATE @ IF COMPILE 2+ COMPILE !
  ELSE 2+ ! THEN ;       IMMEDIATE

: (ATKIN) ( k cfa pfa...) ROT 1 MAX 2 * + ! ; ( Stores cfa at  )
  ( adr=2k+pfa where k=1,2,...,n   Compiled by ATKIN . )

: ATKIN    ( k cfa...) [COMPILE] ' STATE @ IF COMPILE (ATKIN)
  ELSE (ATKIN) THEN ;      IMMEDIATE
  ( Used in form:  k INSTALL cfa ATKIN xxxx                    )
  ( where xxxx is an execution array defined by CASE: , cfa) -->
  ( is the new word to be installed as element k=1,2,...,n     )

: DUM ;                                                       -->

( NOTE:  McCourt's implementation of the function INSTALL ATKIN)
(        does not work inside a : definition. The above does.  )

   MCASE: , A GENERALIZED EXTENSION OF CASE:

   1. Three types of case stuctures:
            a. POSITIONAL ( default)
            b. KEYED
            c. RANGE
   2. Two structure options for each type:
            a. SINGLE WORD EXECUTION ( default )
            b. HIGH LEVEL HEADERLESS CODE SEQUENCE

( Define DOESPART and [DEF] as Execution arrays to be filled )
( in later )

 CASE:  DOESPART DUM DUM DUM DUM DUM DUM DUM ;   ( 6 Cases )

 CASE:  [DEF]    DUM DUM DUM DUM DUM DUM DUM ;

 : MCASE: ( The generalized case compiler )
   <BUILDS SMUDGE !CSP HERE   ( Leave pfa on stack )
```

(3) Range type

```
MCASE: xxxx RANGE
       [ L1 , Hi , ] cfal
       [ L2 , H2 , ] cfa2
       ....
       [ Ln , Hn , ] cfan
       DEFAULT: cfad ;
```

For this type each of the n entries to the case list consists of a pair of values specifying the upper and lower limits of the range , Li and Hi , followed by the execution element, cfai.

When xxxx is later executed in the form k xxxx , the case cfai will be selected if the condition

Li <= k <= Hi

is found during a search of the list. If not, the default case, cfad , will be executed.

Headerless Code Execution Elements

Instead of specifying the execution elements as previously defined FORTH words, the elements may be specified as a sequence of FORTH words in the form:

```
H: ......seq...... ;H
```

or as

```
DEFAULT: .....seq.... ;
```

where ....seq.... is the sequence of executable FORTH words.

Again we have the three applicable case list types, the default type, position, the keyed type and the range type. Examples of the structure of each of these types is

(1) Positional type

```
MCASE: xxxx
       H: ...seql... ;H
       H: ...seq2... ;H
       ...
       H: ...seqn... ;H
       DEFAULT: ....seqd.... ;
```

(2) Keyed type

```
MCASE: xxxx KEYED
       [ kl , ] H: ...seql... ;H
       [ k2 , ] H: ...seq2... ;H
       ...
       [ kn , ] H: ...seqn... ;H
       DEFAULT; ...seqd.... ;
```

```
                               1 C,   ( Default type = 1 )
                               0 C,   ( Number of cases in list = 0 )
                                 ]    ( Enter compile state for list )
     DOES> DUP C@                     ( Gets type )
           DOESPART ;                 ( Executes appropriate search )
                                                                   -->
   : DEFAULT: ( pfa...)   ( Mandatory word used after caselist in)
     ( an MCASE: definition. Compiles ;S . )
     ?COMP EOL ,  HERE OVER C@                   ( ...pfa adrh type )
     [DEF] ;      IMMEDIATE

 : KEYED ( pfa...pfa) ( Used after MCASE: xxxx to set casetype=3)
   3 OVER C! ;                 IMMEDIATE

 : RANGE   ( pfa ...pfa) ( Used after MCASE: xxxx to set type=5 )
   5 OVER C! ;                 IMMEDIATE

 : N? ( n pfa...n pfa f) ( Checks for valid casecount, n , with )
      ( count in case list with pfa specified. True if valid. )
      OVER OVER 1+ C@         ( ...n pfa n count )
      OVER 1 < >R             ( ...n pfa n count )
      > R> OR 0=  ;                                               -->

 ( POSITIONAL TYPE WITH SINGLE WORD EXECUTION OPTION, TYPE 1 )

  : PSFIND ( n pfa...) ( Type 1 case for DOESPART, finds and )
    ( executes case n or default if n<1 or n>casecount for )
    ( MCASE: list pfa. Similar to IX for CASE:                 )
      N?  IF ( Valid n)  2 + SWAP             ( ...pfa+2  n )
          ELSE DUP C@ >R 6 + SWAP DROP R>  ( ...pfa+6  c )
          THEN 1 - DUP + +                    ( ...pfa+K+2[n-1] )
          @ EXECUTE ;

   : PSDEF ( pfa adrdef...) ( Counts # cases entered and stores )
     ( in casecount at pfa+1 . The address of the default cfa is)
     ( at adrdef = pfa+6+2[n-1] )
    OVER 6 + - 2  /                        ( ...pfa n-1 )
     1+ SWAP 1+ C! ;                                               -->

  1 INSTALL PSFIND ATKIN DOESPART
  1 INSTALL PSDEF ATKIN [DEF]

 ( POSITIONAL TYPE WITH HIGH LEVEL DEF IN LIST, TYPE 2       )

  : 2FIND  ( n pfa...adrn) ( Finds address, adrn , of nth high )
    ( level code sequence. Start at pfa of list. Return default)
    ( code address if n<1 or n>casecount )
    N? 0= IF ( def) >R DROP R 1+  C@ 1+ R> THEN  ( ...n+f pfa )
    SWAP >R 4 + 0 BEGIN   1+                   ( ...pfa+4 1,Save n+f)
         R OVER = 0=                           ( ...pfa+4 1 f )
         WHILE ( count not=n+f )               ( ...pfa+4 count )
           >R 2 - @ 2+ R>                      ( ...adrnxt count )
         REPEAT DROP R> DROP ;
                                                                   -->
   : PHFIND ( n pfa...) ( Find and execute hi code seq n in type)
     ( 2 caselist, pfa . Execute default if out of range. )
     2FIND EXECUTE ;

   : H:  ( pfa...pfa adr1) ( Begins headerless definition in an )
     ( MCASE: word, pfa . Compiles dummy link address, compiles )
     ( colon and leaves address of link to be used by ;H  )
     DUP 1+ C@ 1+ OVER 1+ C!             ( Updates casecount )
     DUP C@ 2 MOD IF ( odd)
     DUP C@ 1+ OVER C! THEN              ( Updates type )
     HERE EOL ,                          ( Temporary link )
     COLON , ;                  IMMEDIATE

   : ;H  ( pfa adr1...pfa) ( Terminates headerless definition )
         ( begun by H: . Adjusts link, compiles ;S )
     HERE 2+ SWAP ! EOL , ; IMMEDIATE                              -->

   : PHDEF ( pfa adrdef...) ( Begins headerless defin of )
     ( default. Compiles COLON ) DROP DROP COLON , ;

    2 INSTALL PHFIND ATKIN DOESPART
    2 INSTALL PHDEF ATKIN [DEF]
```

(3) Range type

```
MCASE: xxxx RANGE
   [ L1 , Hi , ] H: ...seq1... ;H
   [ L2 , H2 , ] H: ...seq1... ;H
   ...
   [ Ln , Hn , ] H: ...seqn... ;H
   DEFAULT: ...seqd... ;
```

The interpretation of k in case selecting is the same as previously discussed for the single word execution of the same case type. The only difference is that a FORTH sequence, ...seqi... is executed instead of a single FORTH word, cfai .

Examples

Examples of all 6 possible combinations of case structures are given on Screens 180 and 181. If the screen is loaded and examples tested, typical execution results should be:

EXECUTE	RESULT TYPED
1 BARN	MOO
2 BARN	OINK
18 BARN	PEEP (Default)
1 ZOO	PEEP PEEP PEEP
3 ZOO	PEEP PEEP MOO
-6 ZOO	OINK OINK OINK (Default)
1 FARM	OINK (Default)
77 FARM	MOO
-10 CASE	MOOOINK PEEP (Default)
77 CASE	MOOoooOOO
-10 CORRAL	PEEP PEEP
-1 CORRAL	OINK OINK
309 CORRAL	PEEP OINK MOO
310 CORRAL	MOO (Default)

COMMENTS

1. Kim Harris' case compiler, CASE: avoids the use of OVER = IF DROP ELSE...THEN for every case as used in many of the other CASE constructs. The result is shorter compiled code in the application. The compiler, MCASE: presented here is an extension of CASE: and consequently shares this feature.

2. The compiler, CASE: and the Execution Array introduced by M.A. McCourt in FD II/4 pp 109 are functionally equivalent. Further, the Execution Variable, XEQVAR , of McCourt turns out to be a degenerate case of CASE: with only one element in the case list. The definitions

```
: XEQARRAY CASE: ;
; XEQVAR <LIST DOES> @ EXECUTE ;
```

```

( KEYED TYPE WITH SINGLE WORD EXECUTION OPTION, TYPE 3    )

 : KSDEF ( pfa adrdef...) ( Counts # cases entered and stores )
   ( in casecount at pfa+1. Address of default cfa is    )
   ( adrdef=pfa+6+4[n-1]  )
   OVER 6 + - 4 / 1+ SWAP 1+ C! ;

 : KSFIND ( K pfa...) ( Searches type 3 list for match of key )
   ( to K . Starts at pfa+2 . Executes cfa after matched ) -->
   ( Key or default if no match found. )
   2+ BEGIN 1 >R DUP @ EOL -                ( ...K adr1 f )
            IF ( not EOL) OVER OVER @ = ( ...K adr1 k=@? )
               IF ( matched) 2+             ( ...K adr1+2 )
               ELSE R> 1 - >R 4 + THEN  ( ...K adrnxt )
            ELSE ( EOL ) 2+                 ( ...K adrdef )
            THEN R>                         ( ...K adrnew f )
      UNTIL ( Matched or EOL) SWAP DROP @ EXECUTE ;

  3 INSTALL KSFIND ATKIN DOESPART
  3 INSTALL KSDEF  ATKIN [DEF]

 ( KEYED OPTION WITH HIGH LEVEL DEF IN LIST, TYPE 4      )

 : KHFIND ( K pfa...) ( Searches type 4 list for match of key )
   ( to K . Starts at pfa+2 . Executes high level sequence) -->
   ( following match or default sequence if no match found. )
   2+ BEGIN 1 >R DUP @ EOL -                ( ...K adr1 f )
            IF ( not EOL) OVER OVER @ = ( ...K adr1 k=@? )
               IF ( matched) 4 +            ( ...k adr1+4 )
               ELSE R> 1 - >R 2+ @ THEN ( ...k adrnxt )
            ELSE ( EOL)  2+                 ( ...K adrdef )
            THEN R>                         ( ...K adrnew f )
      UNTIL ( Matched or EOL) SWAP DROP EXECUTE ;

  4 INSTALL KHFIND ATKIN DOESPART
  4 INSTALL PHDEF ATKIN [DEF]  ( Same as type 2 )

 ( RANGE TYPE WITH SINGLE WORD EXECUTION OPTION, TYPE 5   )

 : RSDEF ( pfa adrdef...) ( adrdef= pfa+6+6[n-1] Compute n and)
   ( store at pfa+1 )                                          -->
   OVER 6 + - 6 / 1+ SWAP 1+ C! ;

 : RANGE? ( K adr...f) ( True if K>= value at adr AND K<= value)
   ( at adr+2 )
   >R DUP R @ < SWAP R> 2+ @ > OR 0= ;

 : RSFIND ( K pfa...) ( Searches type 5 list for first occurren)
   ( ce of K within pair of range values. Executes cfa follow- )
   ( ing pair. Executes default cfa if not found )
   2+ BEGIN 1 >R DUP @ EOL -                ( ...K adr1 f )
            IF ( not EOL) OVER OVER RANGE?     ( ...K adr1 s )
               IF ( in range) 4 +           ( ...K adr1+4 )
               ELSE R> 1 - >R 6 + THEN   ( ...K adrnxt )
            ELSE ( EOL) 2+                  ( ...K adrdef )
            THEN R>                         ( ...K adrnew f )
      UNTIL ( In range or EOL) SWAP DROP @ EXECUTE ;       -->

   5 INSTALL RSFIND ATKIN DOESPART
   5 INSTALL RSDEF  ATKIN [DEF]

 ( RANGE OPTION WITH HIGH LEVEL DEF IN LIST, TYPE 6     )

 : RHFIND ( K pfa...) ( Searches type 6 list for first occurr-)
   ( ence of K within pair of range values. If found, executes)
   ( following high level sequence, else executes def sequence)
   2+ BEGIN 1 >R DUP @ EOL -                ( ...K adr1 f )
            IF ( not EOL) OVER OVER RANGE?     ( ...K adr1 s )
               IF ( in range) 6 +           ( ...K adr1+6 )
               ELSE R> 1 - >R 4 + @ THEN ( ...K adrnxt )
            ELSE ( EOL) 2+                  ( ...K adrdef )
```

are fig-FORTH functional equivalents of McCourt's definitions. Hence CASE: can be used as an Execution Array as suggested by McCourt. The definitions of AT , ATKIN and INSTALL on screens 167 and 168 can be used ala McCourt to change the elements in CASE: list words. They are used in the form

k INSTAL yyyy ATKIN xxxx

to change the k'th element in a case list, xxxx defined by CASE: to the code field address of yyyy . Now whenever k xxxx is encountered, the word yyyy will be executed rather than the original word in the k'th position of the case list.

Using the previous CASE: example of BARN , if we execute

2 INSTALL COW ATKIN BARN

the second case in BARN will be changed from PIG to COW. Later execution of 2 BARN anywhere in the program will then type MooOOoo instead of Oink.

Although this is non-structured programming, it is still a valuable programming tool when used properly. The present definitions of INSTALL and ATKIN can be used within a colon definition.

Please note that the use of the Execution Array in the development of MCASE: on screen 169 is purely stylistic. It is not a necessary feature of the development.

3. The essentially unique feature of FORTH is that it is extendable by the user. With an expanding FORTH literature, it is clear to this author that FORTH will improve with time faster than all other languages and that there is no upper limit to its improvement. It has been less than 18 months since I first got FORTH up and running. In that short period of time, thanks to the fig literature, the FORTH system I have running now is, in my opinion, vastly superior to any other language I have ever seen. And it will get better!

```
                THEN R>                              ( ...K adrnew f )
          UNTIL ( In range or EOL ) SWAP DROP EXECUTE ;            -->

  6 INSTALL RHFIND ATKIN DOESPART
   6 INSTALL PHDEF  ATKIN [DEF]  ( Same as types 2 and 4 )
 ;S












 ( MCASE: EXAMPLES )

 : PIG ." OINK " ;
 : COW ." MOO" ;
 : CHICK ." PEEP " ;

 MCASE: BARN  COW PIG CHICK DEFAULT: CHICK ;

 MCASE: ZOO  H: CHICK CHICK CHICK ;H
             H: COW ." oooOOO"    ;H
             H: CHICK CHICK COW   ;H
             DEFAULT: PIG PIG PIG ;

 MCASE: FARM  KEYED  [ 80 , ]  PIG
                     [ 77 , ]  COW
                     [ 67 , ]  CHICK                          -->
                     DEFAULT: PIG ;

 MCASE: CASE  KEYED  [ 77 , ]  H: COW  ." oooOOO"  ;H
                     [ 80 , ]  H: PIG PIG          ;H
                     [ 67 , ]  H: CHICK CHICK      ;H
                      DEFAULT: COW PIG CHICK ;

 MCASE: PEN   RANGE  [ -32768 ,    -1 , ]  CHICK
                     [      0 ,     1 , ]  PIG
                     [      1 , 32767 , ]  COW
              DEFAULT: ;

 MCASE: CORRAL RANGE [ -10 ,  -5 , ]  H: CHICK CHICK    ;H
                     [  -1 ,  -1 , ]  H: PIG SPACE PIG  ;H
                     [   0 ,   0 , ]  H: COW SPACE COW  ;H
  [ 1 , 309 , ] H: CHICK PIG COW ;H  DEFAULT: COW ;                ;S
```

FORTH STANDARDS TEAM MEETING

A FORTH Standards Team meeting will be held in Bethesda, MD, from May 11 through May 14. The meeting is open to the current Standards Team members and a limited number of observers. The site will be the National 4H Center, a self-contained educational facility, just outside Washington, DC. The campus-like Center has meeting rooms, dining facilities and dormitory accommodations.

This four-day meeting will allow world-wide Team members to consider proposals and corrections for the current FORTH Standard and develop future standards policy. Participation is possible by submittal and attendance. Written submittals received by April 30 will be distributed to attendees before the meeting. Late receipts will be distribued at the team meeting. Those wishing to attend must apply without delay, as space is severely limited.

Applicants (other than team members) must submit a biography by April 15 for consideration by the credentials committee. You should include:

1. Your skills and comprehension of multiple FORTH dialects and their application.
2. Why your views are representative of a significant portion of the FORTH community.

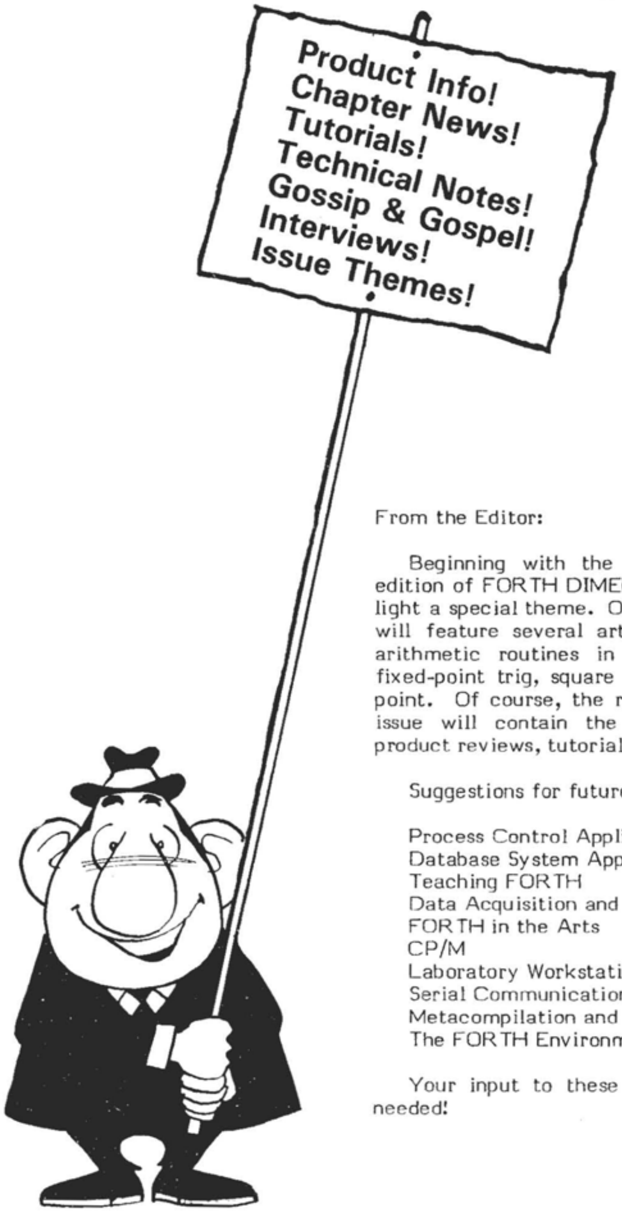
Accommodations are $41 to $47 per day, per person, including meals. Send a refundable $100 deposit (and biography for observers) to the meeting coordinator. You will receive further details on choices in housing and meals.

Submittals are essential if Team actions are to represent the broadening scope of FORTH users. Specific consideration will be given to an addendum correcting FORTH-79, the Team Charter, and alliance with other standards groups. Those not attending may receive copies of submittals by sending $30 to the meeting coordinator.

All submittals and reservations should be directed to the meeting coordinator:

Pam Totta
Creative Solutions
4801 Randolph Road
Rockville, MD 20852
(301) 984-0262

FORTH DIMENSIONS VOLUME IV BEGINS NEXT ISSUE



From the Editor:

Beginning with the next issue, each edition of FORTH DIMENSIONS will highlight a special theme. Our May/June issue will feature several articles on complex arithmetic routines in FORTH such as fixed-point trig, square root, and floating point. Of course, the remainder of each issue will contain the usual technotes, product reviews, tutorials, letters, etc.

Suggestions for future themes include:

Process Control Applications
Database System Applications
Teaching FORTH
Data Acquisition and Analysis
FORTH in the Arts
CP/M
Laboratory Workstations
Serial Communications
Metacompilation and its Alternatives
The FORTH Environment

Your input to these topics is greatly needed!

RENEW TODAY!

LECTURES ON APPLIED FORTH

a two day seminar on Forth and its application

and the

1982 ROCHESTER FORTH CONFERENCE ON DATA BASES AND PROCESS CONTROL

May 17 through May 21, 1982
University of Rochester   Rochester, New York

As part of the 1982 Rochester FORTH Conference on Data Bases and Process Control there will be a two day seminar on Applied FORTH. Managers and programmers will find these lectures very useful for exploring FORTH applications and programming concepts. Each lecturer will also lead a Working Group at the subsequent Conference. Participants should have a copy of Leo Brodie's book, Starting FORTH, which is available from Mountain View Press, PO Box 4656, Mt. View, CA 94040 for $16.00.

Lecturers for the two day seminar are:

Leo Brodie, author of Starting FORTH, on "Beginning FORTH".

Kim Harris, of Laxen & Harris, Inc., on "FORTH Programming Style".

Hans Nieuwenhuijzen, of the University of Utrecht, on "FORTH Programming Environment".

Larry Forsley, of the Laboratory for Laser Energetics, on "Extensible Control and Data Structure".

David Beers of Aregon Systems, Inc., on "A Large Programming Project Case Study: Building a Relational Database in FORTH".

Steven Marcus of Kitt Peak National Observatory, on "Assemblers & Cross Assemblers".

James Harwood of the Institute for Astronomy at the University of Hawaii, on "Computation Tradeoffs".

Roger Stapleton of St. Andrews Observatory, Scotland, on "Hardware Control with FORTH".

Raymond Dessey of Virginia Polytechnic Institute, on "Concurrency, Networking and Instrument Control".

REGISTRATION FORM
(must be received by April 23, 1982)

Name____

Address____

City_____________________State____ZIP___________

Phone (Days) (______) ___________________________

CHOICES TO BE MADE

___ Applied FORTH Seminar, May 17 & 18	$200.00
___ 1982 Rochester FORTH Conference, May 19-21	100.00
___ Housing for: (circle dates) May 16 17 18 19 20 21	$ 13.00/person dbl 16.50/person sgl

TOTAL AMOUNT ENCLOSED $_________

Make checks payable to: "University of Rochester/FORTH Conference"

Send check and Registration to:
Mrs. B. Rueckert, Lab for Laser Energetics, 250 E River Rd, Rochester, NY 14623
For information call: Barbara Rueckert (716) 275-2357

NEW PRODUCTS

Marx FORTH for Northstar
now Available

Marx FORTH is a fast, powerful FORTH system written in Z-80 code. Package includes self-compiler, complete source code, screen editor, and "smart" assembler. Some of the features include calls to the N* directory functions allowing creation, deletion and listing of directories and ease of writing FORTH programs that operate on files created by N* BASIC. Some of the performance features include very fast compile speeds, very fast math, 31-character variable length names, case compiler security, arguments-results, link field in front of name, and many machine code definitions for high speed.

The self-compiler allows you to change anything. If you don't like how I do it, change it! Add anything you want. Price is $85 on N* single density diskette. Source listing available separately for $25.

Perkel Software Systems
1636 N. Sherman
Springfield, MO 65803
(417) 862-9830

FORTH Programming Aids

FORTH Programming Aids are high level FORTH routines which enhance the development and debugging of FORTH programs and complement cross compiler and meta compiler operations with the following features:

- A command to decompile high level FORTH words from RAM into structured FORTH source code including structure control words. This command is useful to examine the actual source code of a FORTH word, or to obtAIn variations of FORTH words by decompiling to disk, editing, and recompiling the modified source code.

- A command to find words called by a specified word to all nesting levels.

- Commands to patch improvements into compiled words and to merge infrequently called words for increased program speed.

- Complete source code and 40-page manual are provided.

Requires a FORTH nucleus using the fig-FORTH model; a minimum of 3K bytes and a recommended 13K bytes of free dictionary space. $150 single CPU license; $25 for manual alone (credit applied toward program purchase). California residents add 6.5% tax. Add $15 for foreign air shipments. Available on 8-inch ss/sd disks (FORTH screens or CP/M 2.2 file of screens), and Apple 3.2 and 3.3 disks; inquite about other formats.

Ben Curry
Curry Associates
PO Box 11324
Palo Alto, CA 94306

New Book: Introduction to FORTH

Introduction to FORTH, a 142-page textbook by Ken Knecht, presents the most complete information available on the MMS FORTH version of the FORTH language. It is written for anyone who wants to learn how to write computer software using FORTH.

No previous knowledge of FORTH is required, but some exposure to Microsoft Level II BASIC will be helpful. Although the book is designed specifically for the MMSFORTH version of FORTH for the Radio Shack TRS-80 Models I and III, most program examples can be adapted to run on other microcomputers that use different versions of FORTH.

FORTH for Ohio Scientific

We've received from Technical Products Co. a copy of their newsletter. This issue contains product news and update screens for FORTH-79. We applaud their intent of good customer support, but note technical errors in definition of several standard words (WORD , R@ , END-CODE , 2CONSTANT , DX). This OSI-FORTH operates with Ohio Scientific OS_65D 3.3 operating system release.

Their new address is Technical Products Co., Box 2358, Boone, NC 28607--ed.

MCZ, ZDS, UDS FORTH

FORTH is now running on Zilog MCZ, ZDS, and Multitech UDS microcomputer systems. It has compiler, editor, assembler, text interpreter, and I/O drives for floppy disk, Centronics printer, and RS232 devices.

Assembly source listing is available now for $10. Source code on diskette is $50 (specify MCZ, ZDS, or UDS). User's manual will accompany each order.

Send checks to Thomas Y. Lo, Electrical Engineering Department, Chung Yuan Christian University, Chung Li, Taiwan, Republic of China.

Software for OSI C1P

Shoot The Teacher - Find the teacher and shoot him with your water pistol. (Teaches basic graphing) $6.95

Speedo Math - Race the computer with your car. (Drills basic addition and multiplication) $6.95

Kamakaze Education Pack - Four programs in one. Addition, X Tables, Spelling, and Place Value Drill. Answer a question and your men go on their last mission. $11.95.

That's Crazy - A takeoff from a famous TV Show where you risk your life to jump over cars and a canyon. A spelling program that provides hours of entertainment. ! $11.95 (specify grade level)

Want Ads Life Skills - A program that helps slow readers understand the Want Ads. Five levels of difficulty. $7.95

Rescue Ship - Transport injured soldiers to the hospital. But the enemy has covered the ocean with mines. One of them could destroy you.

Addition - $11.95
Subtraction - $11.95
Multiplication - $11.95
(all three on tape - $28.00)

Please include $1.00 to cover postage and handling and send to:

Henry Svec
668 Sherene Terrace
London Ontario Canada
N6H 3K1

FORTH VENDORS

The following vendors have versions of FORTH available or are consultants. (FIG makes no judgment on any products.)

ALPHA MICRO

Professional Management Services
724 Arastradero Rd. #109
Palo Alto, CA 94306
(408) 252-2218

Sierra Computer Co.
617 Mark NE
Albuquerque, NM 87123

APPLE

IDPC Company
P. O. Box 11594
Philadelphia, PA 19116
(215) 676-3235

IUS (Cap'n Software)
281 Arlington Avenue
Berkeley, CA 94704
(415) 525-9452

George Lyons
280 Henderson St.
Jersey City, NJ 07302
(201) 451-2905

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

CROSS COMPILERS

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

polyFORTH

FORTH, Inc.
2309 Pacific Coast Hwy.
Hermosa Beach, CA 90254
(213) 372-8493

LYNX
3301 Ocean Park #301
Santa Monica, CA 90405
(213) 450-2466

M & B Design
820 Sweetbay Drive
Sunnyvale, CA 94086

Micropolis

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

North Star

The Software Works, Inc.
P. O. Box 4386
Mountain View, CA 94040
(408) 736-4938

PDP-11

Laboratory Software Systems, Inc.
3634 Mandeville Canyon Rd.
Los Angeles, CA 90049
(213) 472-6995

OSI

Consumer Computers
8907 LaMesa Blvd.
LaMesa, CA 92041
(714) 698-8088

Software Federation
44 University Dr.
Arlington Heights, IL 60004
(312) 259-1355

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

Tom Zimmer
292 Falcato Dr.
Milpitas, CA 95035

1802

FSS
P. O. Box 8403
Austin, TX 78712
(512) 477-2207

6800 & 6809

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

TRS-80

The Micro Works (Color Computer)
P. O. Box 1110
Del Mar, CA 92014
(714) 942-2400

Miller Microcomputer Services
61 Lake Shore Rd.
Natick, MA 01760
(617) 653-6136

The Software Farm
P. O. Box 2304
Reston, VA 22090

Sirius Systems
7528 Oak Ridge Hwy.
Knoxville, TN 37921
(615) 693-6583

6502

Eric C. Rehnke
540 S. Ranch View Circle #61
Anaheim Hills, CA 92087

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

8080/Z80/CP/M

Laboratory Microsystems
4147 Beethoven St.
Los Angeles, CA 90066
(213) 390-9292

Timin Engineering Co.
9575 Genesse Ave. #E-2
San Diego, CA 92121
(714) 455-9008

Application Packages

InnoSys
2150 Shattuck Avenue
Berkeley, CA 94704
(415) 843-8114

Decision Resources Corp.
28203 Ridgefern Ct.
Rancho Palo Verde, CA 90274
(213) 377-3533

68000

Emperical Res. Grp.
P. O. Box 1176
Milton, WA 98354
(206) 631-4855

Firmware, Boards and Machines

Datricon
7911 NE 33rd Dr.
Portland, OR 97211
(503) 284-8277

Forward Technology
2595 Martin Avenue
Santa Clara, CA 95050
(408) 293-8993

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

Zendex Corp.
6398 Dougherty Rd.
Dublin, CA 94566

Variety of FORTH Products

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

Mountain View Press
P. O. Box 4656
Mountain View, CA 94040
(415) 961-4103

Supersoft Associates
P.O. Box 1628
Champaign, IL 61820
(217) 359-2112

Consultants

Creative Solutions, Inc.
4801 Randolph Rd.
Rockville, MD 20852

Dave Boulton
581 Oakridge Dr.
Redwood City, CA 94062
(415) 368-3257

Leo Brodie
9720 Baden Avenue
Chatsworth, CA 91311
(213) 998-8302

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

Inner Access
517K Marine View
Belmont, CA 94002
(415) 591-8295

Laxen & Harris, Inc.
24301 Southland Drive, #303
Hayward, CA 94545
(415) 887-2894

Microsystems, Inc.
2500 E. Foothill Blvd., #102
Pasadena, CA 91107
(213) 577-1471

VENDORS: FORTH DIMENSIONS will go to a product matrix in Volume IV. Send in a list of your products and services by April 18

FIG CHAPTERS

How to form a FIG Chapter:

1. You decide on a time and place for the first meeting in your area. (Allow at least 8 weeks for steps 2 and 3.)

2. Send FIG a meeting announcement on one side of 8-1/2 x 11 paper (one copy is enough). Also send list of ZIP numbers that you want mailed to (use first three digits if it works for you).

3. FIG will print, address and mail to members with the ZIP's you want from San Carlos, CA.

4. When you've had your first meeting with 5 or more attendees then FIG will provide you with names in your area. You have to tell us when you have 5 or more.

Northern California
4th Sat FIG Monthly Meeting, 1:00 p.m., at Southland Shopping Ctr., Hayward, CA. FORML Workshop at 10:00 am.

Southern California
Los Angeles
4th Sat FIG Meeting, 11:00 a.m., Allstate Savings, 8800 So. Sepulveda, L.A. Philip Wasson, (213) 649-1428.

Orange County
3rd Sat FIG Meeting, 12:00 noon, Fullerton Savings, 18020 Brockhorst, Fountain Valley, CA. (714) 896-2016.

San Diego
Thur FIG Meeting, 12:00 noon. Guy Kelly, (714) 268-3100, x 4784 for site.

Northwest
Seattle Chuck Pliske or Dwight Vandenburg, (206) 542-7611.

New England
Boston
1st Wed FIG Meeting, 7:00 p.m., Mitre Corp., Cafeteria, Bedford, MA. Bob Demrow, (617) 389-6400, x198.

Boston
3rd Wed MMSFORTH Users Group, 7:00 p.m., Cochituate, MA. Dick Miller, (617) 653-6136 for site.

Southwest
Phoenix Peter Bates at (602) 996-8398.

Tulsa
3rd Tues FIG Meeting, 7:30 p.m., The Computer Store, 4343 So. Peoria, Tulsa, OK. Bob Giles, (918) 599-9304 or Art Gorski, (918) 743-0113.

Austin John Hastings, (512) 327-5864.

Dallas
Ft. Worth
4th Thur FIG Meeting, 7:00 p.m., Software Automation, 1005 Business Parkway, Richardson, TX. Marvin Elder, (214) 231-9142 or Bill Drissel (214) 264-9680.

Mountain West
Salt Lake City
Bill Haygood, (801) 942-8000

Mid Atlantic
Potomac Joel Shprentz, (703) 437-9218.

New Jersey George Lyons (201) 451-2905.

New York Tom Jung, (212) 746-4062.

Midwest
Detroit Dean Vieau, (313) 493-5105.

Minnesota
1st Mon FIG Meeting. Mark Abbott (days), (612) 854-8776 or Fred Olson, (612) 588-9532. Call for meeting place or write to: MNFIG, 1156 Lincoln Avenue, St. Paul, MN 55105.

Foreign
Australia Lance Collins (03) 292600.

England FORTH Interest Group, c/o 38, Worsley Road, Frimley, Camberley, Surrey, GU16 5AU, England

Japan FORTH Interest Group, Baba-bldg. 8F, 3-23-8, Nishi-Shimbashi, Minato-ku, Toyko, 105 Japan.

Canada - Quebec
Gilles Paillard, (418) 871-1960 or 643-2561.

W. Germany Wolf Gervert, Roter Hahn 29, D-2 Hamburg 72, West Germany,(040) 644-3985.

SIGN UP A FRIEND

START A FIG CHAPTER

PUT THE ORDER FORM ON THE BULLETIN BOARD

FORTH INTEREST GROUP MAIL ORDER

	USA	FOREIGN AIR
□ Membership in FORTH INTEREST GROUP and Volume IV of FORTH DIMENSIONS (6 issues)	$15	$27
□ Volume III of FORTH DIMENSIONS (6 issues)	15	18
□ Volume II of FORTH DIMENSIONS (6 issues)	15	18
□ Volume I of FORTH DIMENSIONS (6 issues)	15	18
□ fig-FORTH Installation Manual, containing the language model of fig-FORTH, a complete glossary, memory map and installation instructions	15	18
□ Assembly Language Source Listing of fig-FORTH for specific CPU's and machines. The above manual is required for installation. Check appropriate boxes. **Price per each.** □ 1802 □ 6502 □ 6800 □ 6809 □ 8080 □ 8086/8088 □ 9900 □ APPLE II □ PACE □ NOVA □ PDP-11 □ ALPHA MICRO	15	18
□ "Starting FORTH" by Brodie. BEST book on FORTH. (Paperback)	16	20
□ "Starting FORTH" by Brodie. (Hard Cover)	20	25
□ PROCEEDINGS 1980 FORML (FORTH Modification Lab) Conference	25	35
□ PROCEEDINGS 1981 FORTH University of Rochester Conference	25	35
□ PROCEEDINGS 1981 FORML Conference, Both Volumes	40	55
□ Volume I, Language Structure	25	35
□ Volume II, Systems and Applications	25	35
□ FORTH-79 Standard, a publication of the FORTH Standards Team	15	18
□ Kitt Peak Primer, by Stevens. An indepth self-study primer	25	35
□ BYTE Magazine Reprints of FORTH articles, 8/80 to 4/81	5	10
□ FIG T-shirts: □ Small □ Medium □ Large □ X-Large	10	12
□ Poster, Aug. 1980 BYTE cover, 16 x 22"	3	5
□ FORTH Programmer Reference Card. If ordered separately, send a stamped, addressed envelope.	FREE	
TOTAL	$______	

NAME__________________________ MAIL STOP/APT______________

ORGANIZATION__________________________(if company address)

ADDRESS__________________________________

CITY______________ STATE______ ZIP______ COUNTRY______________

VISA #______________ MASTERCARD #______________

EXPIRATION DATE ______________ (Minimum of $10.00 on charge cards)

Make check or money order in US Funds on US bank, payable to: **FIG.** All prices include postage. **No purchase orders without check.** California residents add sales tax.

ORDER PHONE NUMBER: (415) 962-8653

FORTH INTEREST GROUP PO BOX 1105 SAN CARLOS, CA 94070

RENEW TODAY!

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

TIME DATED MATERIAL
DELIVER BEFORE
APRIL 15