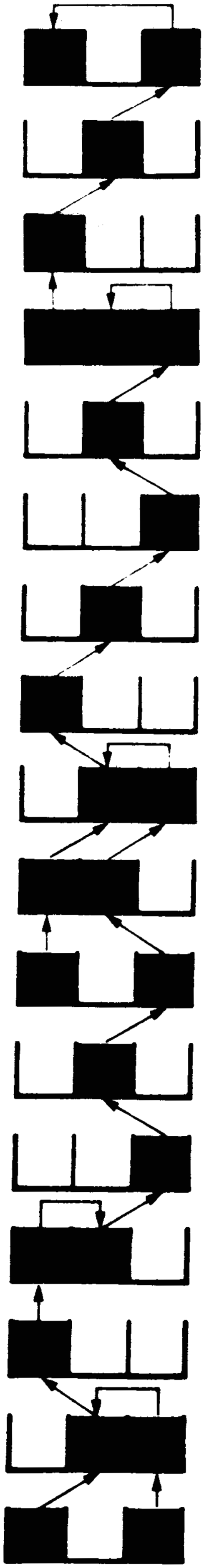


FORTH DIMENSIONS

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

VOLUME II
Numbers 1 - 6

FORTH DIMENSIONS INDEX
VOLUME I, II, 111

INDEX COMPILED COURTESY OF
M. TASSANO
936 DELAWARE WAY
LIVERMORE, CA 94550

TITLE	VOL,PAGE
ADDING MODULES, STRUCTURED PROGRAMMING	II,132
APPLE-4TH CASE	II,62
ARTIFICIAL LINGUISTICS	III,138
ASSEMBLER, 6502	III,143
ASSEMBLER, 8080	III,180
BALANCED TREE DELETION IN FASL	II,96
BASIC COMPILER REVISITED	III,175
BEGINNER'S STUMBLING BLOCK	II,23
BENCHMARK, PROJECT	II,112
BOOK REVIEW, STARTING FORTH	III,76
BRINGING UP 8080	III,40
:CASE	II,41
CASE AND PROD CONSTRUCTS	II,53
CASE AS A DEFINING WORD	III,189
CASE AUGMENTED	III,187
CASE CONTEST STATEMENT	II,73
CASE IMPLEMENTATION	II,60
CASE STATEMENT	II,55
CASE STATEMENT	II,81
CASE STATEMENT	II,82
CASE STATEMENT	II,84
CASE STATEMENT	II,87

INDEX CONTINUED

INDEX CONTINUED

INDEX CONTINUED

INDEX CONTINUED

INDEX CONTINUED

INDIVIDUAL WORDS DEFINED

WORDS CONTINUED

MISC. ENTRIES

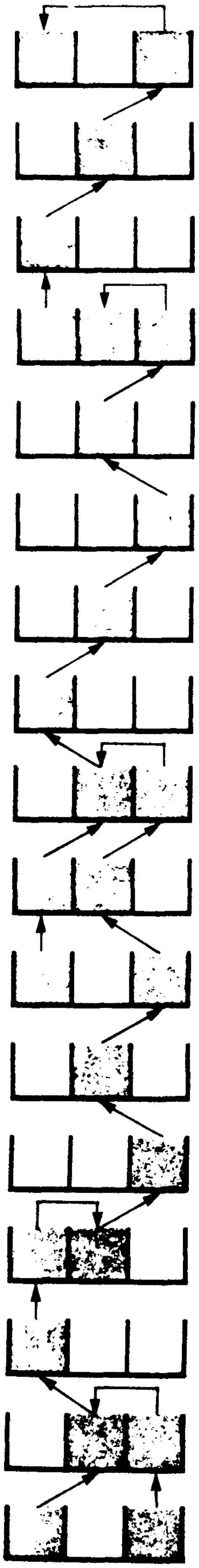
MISC. CONTINUED

INDEX CONTINUED

SCREENS OF FORTH CODE

SCREENS CONTINUED

SCREENS CONTINUED



FORTH DIMENSIONS

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

Volume II
Number 1
Price $2.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No. 1 May/June 1980

Publisher Roy C. Martens

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL 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 $12.00 per year ($15.00 overseas). 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 for 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, although our membership of 1100 is world-wide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

IMPORTANT DATES

April 26	FIG Monthly Meeting, 1:00 pm, at Liberty House, Hayward, CA. Come to the FORML Workshop at 10:00 am and stay on.
May 20	National FIG Meeting, Disneyland Hotel, Anaheim, CA at the NCC Personal Computing Festival. Dinner in the evening and technical sessions all day. Contact: Jim Flournoy, (408) 779-0848.
May 24	FIG Monthly Meeting, 1:00 pm, at Liberty House, Hayward, CA. Come to the FORML Workshop at 10:00 am and stay on.
June 8-13	American Chemical Society
June 21	So. Cal. FIG Meeting, MSI Data Corp., 300 Fischer Ave., Costa Mesa, CA. Noon.
June 28	FIG Monthly Meeting, 1:00 pm, at Liberty House, Hayward, CA. Come to the FORML Workshop at 10:00 am and stay on.

PUBLISHER'S COLUMN

Don't let your membership in FIG crash. Renew today! It's easy. Just send in your check for $12.00 ($15.00 overseas) and you'll be all set for the next six issues of FORTH DIMENSIONS and the FIG notices. If you are in doubt as to whether your membership is up, just look at the address label. If it reads "Renew March 1980" then its time to get that check off. Do it today.

The next issue of FORTH DIMENSIONS is going to be super. It will be a technical issue with all the entries submitted in the Case Contest. Make sure that you receive this important issue, renew your membership in FIG today.

This may sound like a hard pitch for your membership but FIG needs you. The only way that we can keep on publishing FORTH DIMENSIONS and spreading the FORTH word is by having your support. In fact, how about getting others to sign up.

Roy Martens

KIM HARRIS COURSE

A five day intensive course on programming with FORTH will be held July 21-25 at Humbolt State University in Arcata, California. The course will cover the FORTH approach to producing computer applications including: (1) analyzing the requirements of a problem, (2) designing a logical solution, and (3) implementing and testing the solution. Topics will include the usage, extension, and internals of the FORTH language, compiler, assembler, virtual machine, multitasking operating system, mass storage virtual memory manager, and file system. Computers will be available for demonstrations and class exercises. The course will be taught by Kim Harris, and Humbolt State University will give 4 units of credit through the office of Continuing Education. Tuition for the course is $112 per student. The text will be "Using FORTH"; copies will be available at the course for $25 each. Housing is available in very nice dormitory rooms for $9 per person per night or at several nearby motels. Cafeteria meals may be purchased individually or at $10.25 per day. For more information and registration materials write, before June 23:

Prof. Ronald Zammit
Physics Department
Humbolt State University
Arcata, California 95521

FORML NEWS

FORML (FORTH Modification Laboratory) is a research group coordinating individual efforts on the technical evolution of FORTH. Workshop meetings are held the fourth Saturday of each month at 10:00 a.m. at the Liberty House, Hayward, CA. (Make a day of it by staying for the FIG meeting in the afternoon.) Working groups determine and document: the objectives (what problems need to be solved), status of topic (what has already been done), the challenges (what has to be done), the methods (the appropriate approach), the list (detailed topics and problems), the specifications (requirements of valid solutions). You can input directly to the technical committees or to FIG Chairman Kim Harris (see Files DBMS). The committees and leaders are:

Committee	Leader
Numeric Extensions & Floating Point	LaFarr Stuart P.O. Box 1418 Sunnyvale, CA 94088 (408) 296-6236
MetaFORTH - Nucleus	Armand Gambera TTI Inc. 555 Del Rey Ave. Sunnyvale, CA 94086 (408) 735-8080
Concurrency, Multitasking, Executive Communication Synchronization	Terry Holmes 808 Coleman, #21 Menlo Park, CA 94025
Strings	John Cassady 11 Miramonte Road Orinda, CA 94563 (415) 254-2398
Documentation	John S. James P. O. Box 348 Berkeley, CA 94701 (415) 526-8815
Graphics	Howard Pearlmutter 1055 Oregon Ave. Palo Alto, CA 94303 (415) 856-0450
Files DBMS	Kim Harris 1055 Oregon Ave. Palo Alto, CA 94303 (415) 856-0450

FORML needs your help. Come to the next meeting!

THIS IS THE BEGINNING!
THE BEGINNING OF FIG TWO!
THE BEGINNING OF FORTH DIMENSIONS II!
IT'S TIME TO RENEW!
RENEW YOUR MEMBERSHIP IN FIG!
RENEW YOUR SUBSCRIPTION TO FORTH DIMENSIONS!
DO IT ALL FOR ONLY $12.00!
DO IT TODAY!
IT'S EASY!
CHECK THE LABEL FOR RENEW DATE!
IF IT READS "Renew Mar. 1980" SEND A CHECK!

SEND IT TO: FIG, P.O. Box 1105, San Carlos, CA 94070 RENEW NOW!

FORTH, for the Motorola 6809

Raymond J. Talbot, Jr.
7209 Stella Link, Suite 112
Houston, Texas 77025

68'FORTH is an implementation of fig-FORTH for the 6809 microprocessor. It is available on 5" disk configured for an SWTPC SS-50 Buss system with SWTPC MF-68 dual 5" disks and the TSC FLEX 9.0 disk operating system, but it is easily modifiable for other systems (write author for information).

The 6809 is a greatly improved version of the Motorola 6800 8-bit microprocessor. It is almost like having a 16-bit microprocessor, since there are several 16-bit instructions. It has two 16-bit index registers X and Y, and a 16-bit accumulator register D which may also be used as two 8-bit registers A and B. There are many addressing modes, including indirect, autodecrement, and autoincrement.

The two hardware stack registers make it ideal for FORTH — it is almost a FORTH machine in silicon. I have implemented FORTH by the following register assignments:

The FORTH variable stack — U stack register

The FORTH return stack — S stack register

The FORTH instruction pointer (IP) — Y index register

The FORTH register W (which points to the machine code being executed) is never stored (to save an instruction which is usually unnecessary), however, upon entry to a word's machine code, that address is in the -- X index register

Inside a word, one may use X and D without bothering to save their values on entry. If one wants a second index register (very handy for something like CMOVE), then one or more of Y, S, or U registers may be saved in memory (or on one of the stacks).

Before listing the code which makes the FORTH machine, let me describe the notation used to make dictionary entries with the TSC assembler MACRO facility:

```
LASTNM   SET     0              initialize last name address to be
                                  zero; this will mark beginning
                                  of dictionary
WORDM    MACRO                  macro called WORDM to make entry
NEXTNM   SET     *              sets NEXTNM equal to present location
                                  which will be first byte of name
         IFC     &4, IMMEDIATE  conditional compilation for IMMEDIATE
                                  words
         FCB     &1+$C0         first byte is no. of char. with sign
                                  and immed. bit
         ELSE
         FCB     &1+$80         no immed. bit
         ENDIF
         IFNC    &1,1           special case of a 1 character word
                                  will skip this
         FCC     /&2/
         ENDIF
         FCB     $80+'&3        last character has sign bit set
         FDB     LASTNM         link to previous word in dict
LASTNM   SET     NEXTNM         reset LASTNM to point to this word
         ENDM                   end of macro
```

A-10

The &n quantities refer to parameter to the MACRO. E.g.,

MACRO 4,BAS,E

will assemble as

84 42 41 53 C5 LI NK

Where LI NK is the link address to the previous entry. This macro coupled with assembly of addresses allows one to write assembly language code that is essentially just colon definitions, e.g., the macro definition of COLON itself below.

Here is the assembly language listing for the portion of the code which defines the 6809 FORTH machine:

```
         WORDM   1,,:,IMMEDIATE
COLON    FDB     DOCOL, QEXEC, SCSP, CURRENT, AT, CONTXT, STORE
         FDB     CREATE, RBRAK, PSCODE
DOCOL    PSHS    Y                 push IP = Y to ret stack = S
         LEAY    2,X               increment Y to first parameter after
                                     CFA in W = X
NEXT     LDX     ,Y++              get W into X and then increment IP = Y
                                     to point to next instruction
NEXTP    JMP     [,X]              jump indirect to code pointed to by
                                     W = X
         WORDM   2,;,S
SEMIS    FDB     *+2
PSEMIS   LDY     ,S++              reset IP = Y to next address (found
                                     by popping from the ret stack = S).
```

A-11

Some arbitrarily chosen examples of the great economy achieved by this use of the stack registers is given by some words shown here (note: depending on location, some of the BRA have to be the long branch instruction LBRA).

```
        WORDM   7,EXECUT,E
EXEC    FDB     *+2
        PULU    X               pull address from var. stack = U
                                  and put into W = X -- one of
                                  very few cases requiring W
        BRA     NEXTP
        WORDM   1,,+
PLUS    FDB     *+2
        PULU    D               get top item from stack into D
                                  accumulator, add second item,
                                  now top of stack
        ADDD    ,U
PUTD    STD     ,U              store sum back on stack
        BRA     NEXT
        WORDM   2,1,+
ONEP    FDB     *+2
        LDD     ,U              get top item into D
        ADDD    #1              add 1
        BRA     PUTD            put back onto D
        WORDM   1,,@
AT      FDB     *+2
        LDD     [,U]            get # pointed to by add. on stack
        BRA     PUTD            replace top of stack with #
        WORDM   1,,!
STORE   FDB     *+2
        PULU    X,D             gets top into D, second into X
        EXG     X,D             exchange
        STD     ,X              store second into location pointed
                                  to by top
        BRA     NEXT
        WORDM   2,>,R
TOR     FDB     *+2
        PULU    D               pull top item from var. stack
        PSHS    D               push onto ret. sack
        BRA     NEXT

A-12
```

These various fragments of 6809 code can be compared with the corresponding 6800 code in the FIG 6800 ASSEMBLY SOURCE LISTING. Specifically, the 6809 NEXT routine takes 14 machine cycles whereas the 6800 NEXT routine takes 38 cycles.

The 68'FORTH implementation for the 6809 is essentially identical with the 6800 fig version except for the machine code for the words defined that way. Many words which are coded (like PLUS) are shorter in 6809 code because of the 16-bit math. For all of the colon-definition-like-words in 6800 fig-FORTH, I just used my WORDM MACRO; that keeps the source file short.

68'FORTH implements the full fig-FORTH vocabulary as given by the May 1979 6800 ASSEMBLY SOURCE LISTING and the fig-FORTH Installation manual. In addition, particular installation dependent code for EMIT, KEY, and disk read and write are given for a 6809 system with all disk I/O being done via the disk sector read/write routines of the TSC FLEX 9.0 disk operating system. FLEX formatted text files may be read or written in lieu of the terminal. (The word READ switches KEY to read a text file, similarly WRITE switches EMIT to write a text file). Consequently, it is possible to communicate data between FORTH and other FLEX programs (Horrors - BASIC even!!).

Another feature of 68' FORTH is something which should be part of any FORTH system which operates under a host DOS — It has a word _ (underscore on some terminals, left arrow on others) which is followed by a text string (delimited by carriage return or double quote). The text string is passed to the DOS command processor. While in 68'FORTH one can do any FLEX command, e.g., CAT (catalog of FLEX files), DELETE, RENAME, etc., by, e.g.

```
_ CAT 1.F (carriage return)
```

to get a catalog of all files on drive 1 with name starting with F. This type of facility is extremely useful for the exchange of data with other types of programs and for using FORTH in time-sharing systems where other people use other languages. For example, at Rice we operate a PDP-11/55 with the UNIX operating system and FORTH functions as just one time-shared process along with many others. As yet our PDP-11 FORTH does not have this _ word, but I plan to include it in order to take advantage of the many extremely useful utility programs which exist in UNIX. In particular, in that environment, we want to be able to transfer data between tapes and disks as background jobs while we are working with files interactively with FORTH. Also, for number crunching work, other languages are more convenient and faster than FORTH, so we plan to implement certain tasks in other languages to be done as background jobs supervised by UNIX while we use FORTH for just those interactive tasks for which it is ideal.

My main reason for pointing out these FORTH connections to another DOS is to encourage the FORTH standards team to think about standard vocabulary words for these links. FORTH grew up and still largely exists as a stand alone operating system. However, already it is used in some places as simply one language in a multi-language time shared system. I know of two places -- here at Rice where we have begun a rudimentary connection between FORTH and UNIX, and at Kitt Peak National Observatory where their CDC 6400 has very elaborate inconnections between FORTH and CDC's SCOPE.

Editors Note -- This is an excellent example of conversion of a FIG assembly listing to another processor. However one more change is in order. NEXT on the 6809 is only 4 bytes long and the code jump to NEXT takes 3 bytes. On processors this powerful, the code for NEXT is repeated, in-line, wherever needed. This costs one byte but saves 3 clock cycles on each interpretive cycle. The time overhead of indirect threaded code is then 12 cycles. Similarly, PUTD should be expanded in line.

Recursion —
The Eight Queens Problem

Jerry LeVan
Dept. of Math Science
Eastern Kentucky University
Richmond, KY 40475

The Eight Queens problem has been often used as a textbook problem in programming, particularly to illustrate recusion. I present here a solution in FORTH.

Recusion is the technique of allowing a procedure (a FORTH word definition) to call itself. This is normally blocked during FORTH compilation, to allow a old word name to be used in a new definition of the same name. For example:

```
: HELP  CR  CR  HELP  CR  CR  ;
```

The new definition of HELP will execute a previous definition, but with two carriage returns before and after. This is a necessary and common capability.

How then to have a word call itself, if not by name? The answer is MYSELF. This word will compile a call to the word in which it is located:

```
:  DEMO

   - - - -

IF  MYSELF  ELSE  ——  THEN  ;
```

In this example, if the test is true at IF, at MYSELF a call to DEMO will occur. This is accomplished by defining MYSELF as IMMEDIATE. At compile time, MYSELF executes and compiles the execution (code field) address of the most recent (actually incomplete) definition in the CURRENT vocabulary. The fig-FORTH definition is:

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

The Four Queen Problem at hand finds the board row and column locations on which eight chess queens would be safe from mutual attack. This example doesn't check for board rotations or reflections, so more answers are printed than necessary.

The output gives the calculation number on which the answer was found and a list of the eight row numbers, column by column on which the queens are located. Now it's your turn to DO-IT.

```
SCR # 57
 (  8 queens   by Jerry LeVan                       WFR-79DEC02 )
 : 2*    DUP  +  ;  ( double a value )

 : MYSELF    ( allow a word to call itself, by recursion )
      LATEST  PFA  CFA  ,  ;  IMMEDIATE

 : IARRAY   ( makes an array of 1's, as given by input )
     <BUILDS  0  DO  1  ,  LOOP
     DOES>  SWAP  2*  +  ;  ( leave address within array )

  8 IARRAY A   ( these form workspace for the solutions )
 16 IARRAY B
 16 IARRAY C
  8 IARRAY X   ( this contains trial solutions )
 -->

```

```
SCR # 58
 (  more 8 queens                                   WFR-79DEC02 )
 : SAFE
     SWAP  OVER  OVER  OVER  OVER
     -  7  +  C  @  >R
     +  B  @  >R
   DROP  A  @  R>  R>  *  *  ;
 : MARK
     SWAP  OVER  OVER  OVER  OVER
     -  7  +  C    0  SWAP  !
     +  B  0  SWAP  !
     DROP  A  0  SWAP  !  ;
 : UNMARK
     SWAP  OVER  OVER  OVER  OVER
     -  7  +  C  1  SWAP  !
     +  B  1  SWAP  !
     DROP  A  1  SWAP  !  ;  -->
```

```
SCR # 59
 (  more 8 queens                                   WFR-79DEC02 )
 0 VARIABLE TRIES
 : PRINTSOL   (  print one solution )
       ." found on try "  TRIES  @  6 .R  8  0
   DO  I  X  @  1+  5  .R  LOOP  CR   ;
 : TRY          8  0         ( search for answers )
   DO 1 TRIES +!  ?TERMINAL  IF  QUIT  THEN  DUP  I  SAFE
     IF  DUP  I  MARK
          DUP  I  SWAP  X  !
          DUP  7  <
            IF  DUP  1+  ?STACK  MYSELF  ELSE  PRINTSOL  THEN
          DUP  I  UNMARK
       THEN
    LOOP  DROP  ;

 : DO-IT    0 TRIES !  0  CR  TRY  ;   ( This runs the problem )
```

A 'TINY' Pseudo-Code

Bill Powell
Sawbridgeworth, Herts
England

There are some interesting speed/memory trade-offs which depend on the pseudo-code adopted in implementing FORTH. The discussion by David Sirag [1] for the PDP-11 shows DTC to be both faster and more compact, but less flexible (?) than ITC which is the de facto standard. But 6502 FORTH (Programma Consultants) appears to use the JSR/RTS structure. This is faster, but must lead to a lot of code since it now takes 3 instead of 2 bytes to reference each low level (CODE) routine in a high level (COLON) definition.

On my 6502, an 8 bit machine, it seems attractive to call the most frequently used FORTH words with a single byte. My 'TINY' code reads the first byte and then shifts it left to write bits 6 and 7 into the sign and carry flags. For codes $80 thru $FF (carry set) a branch is taken to a 2 byte COLON instruction. The even value we now have is used for the Lo byte for the Instruction Counter (IC). The Hi byte is read by the original IC before being saved on the return stack. This is still a two byte p-code which allows us a vast number of Colon definitions. But we no longer need the Code Address, saving 2 bytes. But we must start these entries at even addresses which will cost 1/2 byte on average.

Next the sign bit is tested. If clear, a branch is taken to a routine for low Literals which pushes the numbers 0 thru $3F on the stack. Then this routine drops back into the 'TINY' interpreter. These low literals (0-63) thus compile into fast single byte p-codes. Frequently used Variables can be stored at these memory addresses making this doubly useful, e.g., User Variables.

For codes $40 thru $7F the above branches fail and we drop into a nucleus CODE routine. This is done by a look-up table which costs two bytes per entry just as the Code Address normally does. We can support up to 64 CODE routines this way. Despite the time taken for bit testing this structure works out quite fast because only one byte has to be fetched. Of course we could arrange for more than 64 CODE entries by defining one that gives access to a three byte structure, but 64 should be enough.

The effect of these one byte instructions is to make the body of COLON definitions much smaller. Literals require 1, 2 or 3 bytes instead of 2, 3 or 4 bytes. On the other hand CONSTANTS and VARIABLES will usually require 3 instead of 2 bytes since in TINY they are compiled like Literals. But these words are infrequent in FORTH because parameters are passed on the stack.

		TINY	JSR/RTS	DTC	ITC
1. CODE (NEXT)	cycles	21	12	19	28
	d.bytes	5	1	3	5
	p.bytes	1	3	2	2
2. ': ;'	cycles	100	24	111	105
	d.bytes	1-1/2	1	5	4
	p.bytes	2	3	2	2
3. Storage e.g. CONSTANT	cycles	30 to 48	? to 58	? to 48	? to 63
	d.bytes	1	3	3	2
	p.bytes	1 to 3	3	2	2
4. Literals	cycles	30 to 48	? to 59	49 to 54	? to 48
	p. bytes	1 to 3	4 to 5	2 to 4	2 to 4
5. A Line Lo level	cycles	250	246	284	340
	p.bytes	14	29	24	23
6. A Line Hi level	cycles	****1237	1056	1412	1591****
	p.bytes	17	29	24	23
7. Program Storage	d.bytes	285	155	465	455
	p.bytes	930	1750	1440	1380
	total.bytes	****1215	1895	1905	1835****

Table comparing p-codes: d.bytes = dictionary overhead
p.bytes = length of p-code required

A-17

The table above analyses three forms of overhead:

1. Time overhead in cycles
2. Dictionary building overhead d.bytes
3 P-code required each time the entry is called p.bytes

Sections 1, 2 and 3 analyse FORTH words of type CODE, COLON, CONSTANT, and Section 4 analyses Literals.

In Section 5 we find the time over head to execute a line assumed to contain 6 CODE, 1 Literal, and 2 Storage (CONSTANT) words, as well as the space for its p-code. Some of the numerals have been assumed low.

Section 6 is like Section 5 except that 3 of the CODE words have been replaced by 3 COLON words of the type in Section 5. At this level we can get a good idea of comparative speeds of execution.

Section 7 gives the storage required for a program of 35 CODE, 20 storage, and 60 COLON words (drawing equally from Sections 5 and 6). This does not include the space for actual data nor for the machine code of the nucleus, but does include all p-code and dictionary overheads apart from the headers.

Taking ITC as 100% we see that the performance becomes:

	TINY	JSR/RTS	DTC
Time Overhead	78%	66%	89%
Space Required	66%	103%	104%

The benchmarks are for the 6502, but similar ranking seems likely for other 8 bit micros. Clearly, longer programs will favor TINY even more. On the other hand JSR/RTS may execute even faster than indicated because the nucleus can make freer use of the cpu registers.

An important aspect of FORTH is the access it gives the user to the structure of the language. Therefore I would still like to see ITC remain the preferred form because of its elegence and flexibility. But TINY has much to offer on small 8 bit systems.

[1] Sirag, D: "DTC v/s ITC for FORTH" FORTH DIMENSIONS Vol. 1, No. 3, Oct./Nov. 1978

;s

FORTH in Literature

At the FORTH Convention, October, 1979, Dan Slater gave a short report on an experiment on communication with killer whales. By use of a touch sensitive plate, the orca could learn to physically equate touching a position with a concept or object. Interest was expressed in using the syntax of FORTH to define new items. By this method a man/whale vocabulary can be built.

The evening Charles Moore read a FORTH poem by Ned Conklin. It is loosely based on a classic of English literature.

```
  :    SONG
          SIXPENCE  !
     BEGIN  RYE  @  POCKET  +!  ?FULL  END
     24  0  DO  BLACKBIRD  I  +  @  PIE  +!  LOOP
     BAKE  BEGIN  ?OPENED  END
     SING  DAINTY-DISH  KING  !  SURPRISE  ;
```

A-21

Bill Ragsdale has submitted two more. This is a familiar quotation, with apologies to Browning:

```
:    LOVE
     CR ." How do I love thee?"
     CR ." Let me count the ways."
          1 BEGIN  CR  DUP  .  1+ AGAIN  ;
```

```
:    RHYME
       JACK  DUP  NIMBLE  BE
             DUP  QUICK  BE
       CANDLE-STICK  OVER  JUMP  ;
```

Finally here is an actual, full poem. It is taken from "The Space Childs Mother Goose" by Frederick Winsor, Simon and Schuster, 1958. It consists of eleven stanzas and is almost recursive.

The first two screens compile the primitives from which the poem is recited, by loading of the last screen. The computer's recitation occurs stanza by stanza with the operator indicating his interest and approval by operating any terminal key at the REST after each stanza.

```
SCR # 108
  0 (  The Theory that Jack built                          WFR-79DEC15 )
  1 ( From The Space Child's Mother Goose, Frederick Winsor )
  2 : RECITE       110  LOAD  QUIT  ;   ( say this poem )
  3 : THE                    ." the "   ;
  4 : THAT         CR        ." That "   ;
  5 : THIS         CR        ." This is "  THE   ;
  6 : JACK                   ." Jack built"  ;
  7 : SUMMARY                ." Summary"  ;
  8 : FLAW                   ." Flaw"  ;
  9 : MUMMERY                ." Mummery"  ;
 10 : K                      ." Constant K"  ;
 11 : HAZE                   ." Erudite Verbal Haze"  ;
 12 : PHRASE                 ." Turn of a Plausible Phrase"  ;
 13 : BLUFF                  ." Chaotic Confusion and Bluff"  ;
 14 : STUFF                  ." Cybernetics and Stuff"  ;
 15 : THEORY                 ." Theory "  JACK  ; -->
```

```
SCR # 109
  0 (  More Poem                                           WFR-79DEC15 )
  1 : BUTTON              ." Button to Start the Machine"  ;
  2 : CHILD               ." Space Child with Brow Serene"  ;
  3 : CYBERNETICS    ." Cybernetics and Stuff"  ;
  4 : HIDING        CR  ." Hiding " THE  FLAW  ;
  5 : LAY       THAT  ." lay in "  THE  THEORY  ;
  6 : BASED     CR     ." Based on "  THE  MUMMERY  ;
  7 : SAVED         THAT  ." saved "  THE  SUMMARY  ;
  8 : CLOAK  CR     ." Cloaking "  K  ;
  9 : THICK IF THAT ELSE CR ." And " THEN ." Thickened " THE HAZE ;
 10 : HUNG            THAT  ." hung on " THE  PHRASE     ;
 11 : COVER IF THAT ." covered " ELSE CR ." To cover " THEN BLUFF ;
 12 : MAKE        CR   ." To make with "  THE  CYBERNETICS  ;
 13 : PUSHED      CR   ." Who pushed "  BUTTON  ;
 14 : REST        46  EMIT  10  SPACES  KEY  DROP  CR  CR  CR  ;
 15 : WITHOUT  CR ." Without Confusion, exposing the Bluff" ; RECITE
```

```
SCR # 110
  0 (  Recite our poem                                     WFR-79DEC15 )
  1 CR  CR  CR  THIS  THEORY  REST
  2 THIS  FLAW  LAY  REST
  3 THIS  MUMMERY  HIDING  LAY  REST
  4 THIS  SUMMARY  BASED  HIDING  LAY  REST
  5 THIS  K  SAVED  BASED  HIDING  LAY  REST
  6 THIS  HAZE  CLOAK  SAVED  BASED  HIDING  LAY  REST
  7 THIS PHRASE 1 THICK CLOAK SAVED BASED HIDING LAY REST
  8 THIS BLUFF HUNG 1 THICK CLOAK SAVED BASED HIDING LAY REST
  9 THIS STUFF 1 COVER HUNG 0 THICK CLOAK SAVED BASED HIDING
 10   LAY REST
 11 THIS BUTTON  MAKE 0 COVER  HUNG 0 THICK  CLOAK  SAVED
 12   BASED  HIDING  LAY  REST
 13 THIS CHILD PUSHED CR ." That made with " CYBERNETICS  WITHOUT
 14   HUNG CR ." And, shredding " THE HAZE CLOAK CR ." Wrecked " THE
 15   SUMMARY BASED HIDING CR  ." And Demolished " THEORY REST
```

FORTH, Inc. News

A series of free seminars and paid ($100-125) workshops is being offered; polyFORTH will be presented. The schedule is: Palo Alto, May 8 & 9; Rochester, NY, May 13; Boston, May 14 & 15; New York, June 10; Cherry Hill, June 11; Washington-Baltimore, June 12 & 13; Houston, June 16 & 17; New York, June 18; Palo Alto, June 24 & 25. For more information and/or to register: Call Kris at FORTH, Inc. (213) 372-8493.

FIG DOINGS

Intensive Short Course

The American Chemical Society is offering a number of five day, hands-on, in-depth lab courses on microprocessors and minicomputers. The participants will have access to a PDP-11 network running FORTH. Sessions are June 8-13, September 7-12 and December 14-19 at VPI, Blacksburg, VI at a cost of $485 for ACS members and $550 for non-members. For more information contact ACS Short Courses, 1155 Sixteenth St., N.W., Washington, DC 20036.

FIG GROUPS FORMED OR FORMING

San Diego - Call Guy Kelly (714) 268-3100 ext 4784 or Tom Boyle (714) 571-7711

Seattle - Call Dwight Vandenburg (206) 542-8370 or Terry Dettman (206) 821-5832

Mass. - Third Wednesday at 7:00 pm in Cochituate, MA. Call Dick Miller (617) 653-6136

Virginia - Call Joel Shprentz (703) 437-9218 or Paul van der Eijk (703) 354-7443

Texas - In Houston, call Jeff Lewis (713) 729—3320, in Dallas, call John Earls (214) 661-2928 and in Denton, call Dean Vieau (313) 493-5105

Arizona - Call Dick Wilson (602) 277-6611 ext. 3257

Oregon - Call Ed Kammerer (503) 644-2688

New York - Write Tom Jung, 704 166th St., Whitestone, NY

Michigan - Call Dwayne Gustaus (817) 387-6976

OTHER PUBLICATIONS

Dick Miller has sent the first issue of the MMS FORTH Newsletter. It's jam packed with news, tips and updates for MMS FORTH users on the TRS-80.

It's a service to registered owners, and Dick would be glad to send a sample copy to prospective users. Write Miller Microcomputer Services, 61 Lake Shore Road, Natick, MA 091760.

* * *

Thanks to Fig member Frank Dougherty (325 Beacon Street, Belvedere, IL 61008) for the writeup in the Blackhawk Bit Burners Newsletter. Frank discussed the language and our efforts, as well as the dialect STOIC.

FORTH for Microcomputers by John S. James originally published in Dr Dobbs Number 25 May 1978 has been reprinted first in ACM SIGPLAN NOTICES, Oct. 1978 and now in an IEEE Tutorial: MICROCOMPUTER PROGRAMMING AND SOFTWARE

SUPPORT, IMSONG LEE, EDITOR, IEE cat No. EH0 140-4 to quote from this publication "James gives a compact, but not necessarily easy, introduction to a stack oriented, interactive programming language called FORTH. A better tutorial presentation may be found in the manual, PROGRAM FORTH, A PRIMER, by Gregg Howe, Steward Observatory, University of Arizona, 1973." The current availability of this document is unknown.

More on STOIC-II

Technical Report TR-001

"EDIT79, A STOIC-II Programming Example" (63 pages) $7.00

This report represents and example of a non-trivial program written entirely in STOIC-II. The program, a text editor, was cross-compiled to produce a stand-alone object program, thus facilitating benchmark comparisons with the CP/M editor which it closely resembles. Included in the report are the benchmark results, a brief user's guide, and source code for the editor along with extensive comments.

Contact: Jeff Zurkow
Avocet Systems
804 South State Street
Dover, DE 19901

KIM HARRIS COURSE

A five day intensive course on programming with FORTH will be held July 21-25 at Humbolt State University in Arcata, California. The course will cover the FORTH approach to producing computer applications including: (1) analyzing the requirements of a problem, (2) designing a logical solution, solution. Topics will include the usage, extension, and internals of the FORTH language, compiler, assembler, virtual machine, multitasking operating system, mass storage virtual memory manager, and file system. Computers will be available for demonstrations and class exercises. The course will be taught by Kim Harris, and Humbolt State University will give 4 units of credit through the office of Continuing Education. Tuition for the course is $112 per student. The text will be "Using FORTH"; copies will be available at the course for $25 each. Housing is available in very nice dormitory rooms for $9 per person per night or at several nearby motels. Cafeteria meals may be purchased individually or at $10.25 per day. For more information and registration materials write, before June 23:

Prof. Ronald Zammit
Physics Department
Humbolt State University
Arcata, California 95521

RENEW NOW!

tiny-FORTH

A version of fig-FORTH tailored to the TRS-80, Level II with 16K bytes of memory minimum. I/O devices supported are: keyboard, display and cassette tape recorder. New words can be defined to control other devices. The editor is identical to the fig-FORTH editor and the output format is modified slightly to fit the TRS-80 display. Documentation includes: introduction, editor, graphics, assembler, advanced tiny-FORTH and applications. The style is tutorial with hundreds of examples that teach tiny-FORTH in a hands-on mode. $29.95 for tiny-FORTH cassette and full documentation for the Level II, 16K TRS-80 plus $1.50 for shipping and handling ($5.00 outside the US). The Software Farm, P.O. Box 2304, Reston, VA 22090.

KIM-1 FORTH

This version was written from the FIG model by Ralph Deane of Vancouver, Canada. It contains a complete programming system which has an interpreter/compiler as well as an assembler and editor. All terminal I/O is funnelled through a jump table near the beginning of the software and can easily be changed to jump to user-written I/O drivers. 6502 FORTH resides in 8K of RAM starting at $2000 and can operate with as little as 4K of additional contiguous RAM. $94.00 for 6502 FORTH system on KIM cassette. $16.50 for 6502 FORTH user manual. Eric C. Rehnke, 540 S. Ranch View Circle, #61, Anaheim Hills, CA 92087.

Heath H89 and H8

FORTH for the Heath 89 and 8 is possible with the fig-FORTH 8080, Version 1.1 (as demonstrated by Jim Flournoy at the January FIG meeting). Walter Harris implemented and brought up the code on his dual disc H8 and reassembled it for the H89. For more information, contact: FORTH Power, P.O. Box 2455, San Rafael, CA 94902.

HONEYWELL FORTH SYSTEM

Source Data Systems announces a language for non-programmer data definition, transaction definition, file definition and report generation for Honeywell Level 6 Minis. Designed for information management and retrevial when used together is SDS's Source Data Entry package. For more information, contact: Source Data Systems, 208 2nd Avenue, S.E., Cedar Rapids, IA 52406.

AMD 2901 FORTH PROCESSOR

Functional Automation unleashes the I/O thing, a FORTH based front end processor for its AMD 2901 based 64 bit wide microprogrammed computing engine. The system programming language is FASL (Functional Automation System Language) which is available users. For more information, contact: Functional Automation Inc., 3 Graham Drive, Nashua, NH 03060.

STOIC

STOIC, essentially an extension of FORTH, is a general purpose interactive program, assembler, debugger, loader and operating system within a single consistant architecture. With core efficiency and high running speeds, the language is extremely flexible permitting the user to develop a working vocabulary of subroutines tailored to specific applications.

The entire package, including a library of predefined subroutines, is copyrighted but available to educational users. STOIC requires three discs, two are STOIC itself and the third contains a bootstrap that permits the entry of STOIC through CP/M and the continued use of CP/M disc I/O under STOIC. For more information, contact: Steven Burns, Massachusetts Institute of Technology, Room 20-119, Cambridge, MA 02139.

68'FORTH FOR 6809

68'FORTH is a 6809 implementation of the FORTH Interest Group standard vocabulary of this powerful language.

68'FORTH consists of full FORTH Interest Group standard (May 1979) vocabulary with names to 31 characters, 16 and 32 bit integer math, compiler error checking, and source text editor. System is supplied with additional vocabulary to simulate disk in memory (useful for modifying to work with other disk systems or enabling cassette-only operation), to use disk for virtual memory (allows large data sets to be used in small memory), to interface with FLEX 9.0 text files for input and output, and to perform standard FORTH disk block read and write. System is supplied on 5" floppy disk configured for SWTPC MF-68. Minimum memory requirement is 8k for FLEX plus 12K of user space. Documentation contains description of all vocabulary words, information on configuring for individual system, and basic tutorial for FORTH language. Information is available for reconfiguring to interface with other disk operatings systems.

FLEX 9.0 format 5" disk plus documentation: $39.95.

Talbot Microsystems
7209 Stella Link, Suite 112
Houston, TX 77025

PRODUCT RELEASE

8080 Assembler Available

John Cassady, who did the original fig-FORTH 8080 listing, has now released an 8080 FORTH assembler. John's assembler handled all Intel mnemonics and can easily be altered to Zilog, as it is published as source code. It handles structured assembly conditionals IF, ELSE, THEN, BEGIN, UNTIL, WHILE, and REPEAT. It is integrated with the FIG security package to verify correct structuring of conditionals, during assembly. John provides for named subroutines as well as CODE definitions.

Send $3.25 (includes postage) to John Cassady, 339 15th Street, Oakland, CA 94612.

PolyFORTH-CP/M

polyFORTH-CP/M is FORTH Inc.'s polyFORTH, interfaced to run on nearly any 32K or larger CP/M based system. When loaded, polyFORTH-CP/M finds and links-up to the CP/M I/O drivers, initializes itself, and responds "up" on the system console. At this point, true polyFORTH is running, that is, FORTH structured (screen oriented) diskettes must be used. It is important to realize that polyFORTH-CP/M does not run under CP/M, it runs in place of CP/M, utilizing only the CP/M I/O drivers.

The polyFORTH-CP/M system, as supplied by M&B DESIGN, is a value-added system. FORTH Inc.'s complete 8080 polyFORTH system is supplied, plus an additional diskette and manual containing interface material. Also provided, is a CP/M utility that allows transferring polyFORTH blocks (screens) to a CP/M file, and transferring a CP/M file to polyFORTH blocks. Source is supplied for the entire polyFORTH system, the polyFORTH-CP/M components, and the transfer utility.

polyFORTH-CP/M is available directly from M&B DESIGN for $4,000 on a wide variety of diskette formats. Contact:

M&B DESIGN
820 Sweetbay Dr.
Sunnyvale, CA 94086
(408) 243-0834

FORTH for Poly-88

fig-FORTH for the 8080 as implemented by John Cassady and modified by Kim Harris is now available for the Poly-88.

This version uses cassette and ram simulation of disc, and includes full use of upper and lower case characters as well as the Greek character set, as well as high speed graphics. An editor and an assembler are included.

The complete system price is	$50.00
FORTH on cassette (needs 8K RAM 2000-4000H)	25.00
Poly-88 Forth User's Guide	10.00
8080 fig-FORTH Source Listing	10.00
Installation Manual (F.I.G. Model)	10.00

(CA residents add 6% tax)

Write: Jeff Fox (415) 843-0385
2223 Byron
Berkeley, CA 94702

LOTS OF FORTH

ANCON provides a wide variety of FORTH products, including: Hobby versions; TRS-80 Cassette, $29.95; Heath H8-H89, $49.94; 8080 CP/M 8in, $49.95; 6809 5" Flex, $49.94.

Personel systems; TRS-80 Tape, $45.95; Disc, $65.95; 8080 CP/M 8" $125.00; pdp-11, $140.00; Northstar, Micropolis.

Commercial/Industrial/Scientific versions available for specific requirements. Jim Flournoy, ANCON, 17370 Hawkins Lane, Morgan Hill, CA 95037, (408) 779-0848.

RENEW NOW!

LYON'S DEN

[Editors note — George Lyons has corresponded on FORTH topics over the full life of FORTH DIMENSIONS. He addresses technical and philosophical topics. We're formalizing his contributions into a bylined column. Welcome to the Lyon's Den.]

I suspect that a paramount issue in decisions on whether to use FORTH or another language is a tradeoff between language convenience and compiler convenience. By implementing a complex syntax a PASCAL compiler, say, automates part of the programming task at the expense of a time-consuming source entry and processing operation. Standard FORTH seems to be at the other extreme, leaving more explicit details to be coded by the programmer, for the gain of easier processing with an interactive resident compiler.

The polarization of FORTH and its alternatives on this scale may only be due to the absence of standard FORTH vocabularies to provide the same degree of automation traditional languages supply. I wonder if a quantum jump in FORTH's popularity would result from supplying compilers for traditional languages implemented in FORTH. Possibly, transparently obvious FORTH language features could be provided achieving the same results. The areas where the greatest impact might come are PASCAL data structures, ALGOL procedure argument passing and dynamic local storage allocation, and APL matrix algebra.

If techniques for these operations in FORTH were widely known no one would make the mistake of referring to FORTH as a species of macro assembler. By demonstrating that traditional language convenience is available in FORTH users might be motivated to take advantage of the extensibility of FORTH to go beyond the limitations of the traditional approaches.

March 14, 1970

George B. Lyons
280 Henderson Street
Jersey City, N.J. 07302

Programma was nice enough to supply me with a reassembled version of their Apple-FORTH Kernel plus screens of the dictionary entries for my KIM-1. This was all entered by hand, painfully debugged, editor programs written (in their FORTH), etc. I then tried to duplicate the "PI" routine in the Dr. Dobbs Journal, only to find that Programma didn't carry extra bits of intermediate accuracy in the multiply routine. Then another week of spare time (midnight oil) work to rewrite the math routines to allow "*/MOD" to work properly. It finally worked.

I'm not bitter though. Through all of this I learned enough of FORTH-like programming to be more enthusiastic than ever, but disappointed that the example programs I've received from you are not usable by the Programma version. I am therefore eagerly awaiting the availability of the 6502 version of fig-FORTH for my KIM-1.

Edward J. Bechtel, M.D.
Newport Beach, CA

Should you have any books, manuals or other documents pertaining to FORTH which are available by special order, I would like to have a list. There seems to be a real need for textbooks or tutorials which will carry a user from the most simple FORTH constructions to the very elaborate ones like <BUILDS and DOES>. (See #1 below.)

For your information, I am working with the mmsFORTH implementation from Miller Microcomputing Services. I am quite satisfied with the system to date, and look forward to other extensions. I have distributed several FORTH programs to MMS which they may use in their newsletter. Should the FORTH Interest Group have a program exchange or publish programs, I will submit these programs to you also. (See #2 and #3 below.)

Andrew W. Watson
Vinton, VA

Editor...

#1 - Use the Mail Order page and see Information and New Products sections of FD.

#2 - Send programs to FD!

#3 - Address for Miller Microcomputing Services, 61 Lake Shore Road, Natick, MA 01760.

I want to tell you how impressed I am at the quality of the Installation Manual and the 6800 Assembly Source Listing!

The 6800 listing provided everything I needed to build an identical source file. The Symbol Table and hex dump were especially useful in tracking down the last small typos. (I used the check sums for the 'S1' dump to locate typos such as INS instead of INX.) To get FORTH running on my system, all I did was to modify the ACIA address and delete the coding for Trace.

I notice a peculiar behavior regarding the stack. If I type . when the stack is empty, I get an error message, as expected. But after the error message, there are two numbers on the stack. Is this normal?

Gordon Stallings
Bartlesville, OK

Editor...

The numbers left on the stack after an error are the block number and character offset. See ERROR. This allows WHERE (Scr88) to display the offending text.

Thanks to John James and FIG, I've upgraded my sub-FORTH to what I now call 2650-FOURTH. To date, except for the disk I/O verbs, my FORTH more or less matches Mr. James' FORTH with the exception that I've incorporated an assembler, it's fully ROM based and it has a few more primitives. I do support a cassette I/F but can't use the full power of the fast virtual storage. I will release a copy of my 2650-FORTH to FIG as well as any application work that I've done.

Myself being broadly classified as a computer architect or computer designer, I have a very keen interest in turning out a FORTH engine (to borrow a phrase from Western Digital), and will attempt the implementation. I will probably use the 2900 series bit slices since I have all the development tools. Is there someone in this vein that I could contact?

Edward J. Murray
Pretoria, Union of South Africa

Editor...

Look forward to receiving your 2650-FORTH. Address for John S. James, P.O. Box 348, Berkeley, CA 94701.

I was somewhat disconcerted when I read the article by Mr. David J. Sirag, "DTC Versus ITC for FORTH on the the PDP-11", FORTH Dimensions, Volume 1, No. 3. The author has, I believe, misunderstood the intent of the article by Mr. Dewar.

In Mr. Dewar's ariticle, the definitions of direct threaded code (DTC) and indirect threaded code (ITC) are:

> "DTC involves the generation of code consisting of a linear list of address of a linear list of address of routines to be excuted."

> "ITC..." (involves the generation of code consisting)"... of a linear list of addresses of words which contain addresses of routines to be executed."

As applied to the FORTH type of hierarchial structure (hierarchial indirect threaded code?), I would extend Mr. Dewar's definition to be:

> "ITC involves the generation of code consisting of a linear list of addresses of words which contain addresses of routines to be executed. These routines may themselves be ITC structures."

However, Mr. Sirag based his conclusions on the following loose definition:

> "The distinction between DTC and ITC as applied to FORTH is that in DTC executable machine code is expected as the first word after the definition name; while, in ITC the address of the machine code is expected."

Obviously, the two men are not referring to the same things. Mr. Dewar is referring to the list of addresses which define the FORTH word, while Mr. Sirag is referring to the implementation of the FORTH interpreter. If indeed Mr. Sirag's statement were true (which it is not) that their "analysis contradicts the findings of Dewar", then they should have implemented a DTC language rather than the ITC language of FORTH! Indeed, a careful examination of what is actually occuring in LABFORTH reveals that their techniques are logically identical to Dewar's ITC. They have simply, through clever programming, taken advantage of a particular instruction set and architecture. It is beyond the scope of this letter to prove this equivalence, or to suport the FIG desire to have a common implementation structure for all versions of FIG FORTH.

Please note that I am <u>not</u> quibling over semantics with Mr. Sirag. All definitions are arbituary. (However, the value of a definition lies in its consistency, precision, and useability. I find Mr. Sirag's definition of DTC and ITC to be inconsistent with the environment in which he operates, FORTH, and thus quite useless.) My intent is two fold: (1) I am a self-appointed defender of the excellent work of Mr. Dewar, and (2) I want to correct any misconceptions concerning this issue for readers of this newsletter who did not have access to Dewar's (better) definition of DTC and ITC.

Jon F. Spencer
Sherman Oaks, CA

Many thanks to John Cassady for writing an excellent 8080 FORTH and to Kim Harris for implementing the necessary mods. I received 8080 fig-FORTH Ver. 1.1 on 2 October 1979 and within a few days had the assembly language source typed in and assembled. A day or two later the editor with a suitable patch for the MATCH code was up and running along with the disk based long error messages. I have been learning and gaining experience with fig-FORTH ever since.

After more than a year of using STOIC from volume #23 of the CP/M Users Group it is really nice to be using a true FORTH that is consistant with the examples in the FORTH Inc. manuals and the several articles that have appeared on FORTH. I cannot over-emphasize how well documented the fig-FORTH system is and how easy the system was to bring up. No bugs or errors have been uncovered in nearly six months of use.

The only thing missing from this otherwise nearly perfect package is the assembler vocabulary. Is an 8080/Z-80 assembler vocabulary available from the FORTH Interest Group or if not is any planned? If an 8080 assembler is available or is planned a short note or a word about future plans in the next issue of FORTH DIMENSIONS would be sufficient.

In any case I hope I get to see some of you at NCC in May so that I can personally thank you for making FORTH available to me...

Sincerely,

M. Paul Farr
2250 Ninth Street
Olivenhain, CA 92024

Editor --

Yes! An 8080 assembler is now available in source code to complement 8080 fig-FORTH. Send $3.25 (includes postage) to John Cassady, 11 Miramonte Road, Orinda, CA 94563.

Many thanks for the fig-FORTH installation manual glossary and FORTH Model, which have been difficult but enjoyable items of study since they arrived a couple of weeks ago.

Like many of your members I became interested in FORTH without having access to a FORTH system, and gained my first practical familiarity by using the FORTH low level interpreter style of linkage on machine code programs.

With help from the Model I have now got to grips with the outer interpreter and virtual memory system, and will be getting together with Bill Powell and other FORTH fanciers over here on an cooperative implementation effort.

Many thanks for your effort and creativity, which are not unappreciated!

Bill Stoddart
15 Croftdown Road
London NW5, England

Editor's note -- Bill had a marginal note to this letter: "certainly grows on you. This really is 'Computer Liberation.' BASIC was just a red herring."

Do you have a Z80 version of fig-FORTH? It is not listed on your order sheet but reading the text I got the impression that you do.

By the way, I have a tutorial paper discussing assembly programming in FORTH environment for both 8080 and Z80. It is available, including source listing written in fig-FORTH, from KALTH microsystems. The price is $5.-US for 8080/85 version, $7.-US for Z80 version or $10.-for both (add 15% in Canadian funds).

Also, I am working on the assembler for the Intel 8086/88. If I knew that there are also other people interested in it, that would motivate me getting it complete sooner. (It is a cross-assembler that can be run on any FORTH based system.)

Yours truly,

Kalman Fejes
KALTH microsystems
P.O. Box 5457, Station "F"
Ottawa, Ont., Canada

Editor --

Fig doesn't have a plan a Z-80 version of fig-FORTH. We would be pleased to publish a contributed version, if as complete as the 8080 Version 1.1

As a participant in the Forth International Standards Team, I cast a yeah vote for the inclusion of "TO" and its requisite definition of VARIABLE (though I prefer the name FIELD). Although I was first exposed to this definition on Catalina Island, it has many similarities to my own implementation of FIELD and RECORD.

In its simplest form, as outlined by Paul Bartholdi, FORTH DIMENSIONS 1/4, integer variables of predetermined precision are defined to behave as bidirectional constants. Normal behavior is to push their stored value onto the stack. A momentary, alternate behavior is to pop the stack value into their confines. This temporary behavior occurs only when referenced after the word "TO", which sets a direction flip-flop. Thus

```
VARIABLE A          VARIABLE B
10 TO A             A TO B
```

will place 10 into A and B without using the @ (fetch) and ! (store) operators.

Each of us, who has implemented a version of "TO", encounters some exasperation in dealing with the addresses appearing on the stack. Since, in the prior illustration, neither A nor B supplied its address for TO's execution we ponder the shortcomings of this newly offered definition and reluctantly sprinkle our procedures with @ and !.

FORTH is an elaboration on the indirect threaded list program architecture. As programmers we are free to add indirection to our methods of accessing and manipulating data. Indirection, however, is only a navagation technique for constructing the address required by the hardware to implement our desired operation. When at the end of our circuitous logic, are we then to complain "What can I do about this address".

Let's face it, @ and ! are perfect operators.

I value TO and its implications in system structure. The procedures written using "TO" are more readable than standard Forth, and result in fewer visits to NEXT as they are executed. "TO" will be included in any system I generate, together with other essential words, which include @ and !.

As a Forth fanatic and a FORTH DIMENSIONS fan I sincerely hope that the newsletter will continue. If there is some assistance I can render please advise.

Williams S. Emery
2700 Peterson Place, #53D
Costa Mesa, CA 92626

Editor --

You're doing it! By thoughtful correspondence and participation in group events people such as Bill are multiplying our efforts.

STRUCTURED VARIABLES

From time to time at the Fig meetings the question of structured variables arises. This is a proposal for how they might be handled.

The December 1978 issue of communications of the ACM contained a paper by John Backus on "Functional

Programming" (also called variable free programming). I believe a variation of his ideas could be implemented in FORTH. Suppose we are given a pair of queues with bases at opposite ends of available memory pointing toward each other. Then enter an array into one of them and begin processing it. Let the results go to the other queue as they are developed. Multiple steps would alternate between the queues until a final result is obtained. These alternating queues can give some of the effects of functional programming (1) large state changes, (2) limited memory of past states, (3) no concern with garbage collection, 4) variables not named or declared.

Backus placed operators within the data. This could be done or not, as experience dictates. These queues are not to replace the stack which FORTH already has. The stack could be used to hold what I would call operator variables or modifiers.

Let us look at a couple of simple examples. Suppose we wanted to transpose an array. 1 2 3
4 5 6
Enter it into one queue. [1 2 3 4 5 6]. Type in the transpose command. 2 1 TP. The 2 and the 1 go on the stack so the transpose function knows what kind of a transpose is desired. The result will come out on the other queue: 1 4 2 5 3 6]

Should we wanted to sum a vector. [1 2 3 4 5 6]. Type in a reduction command. ' + RD. The 'Tick' put the address of 'Plus' on the stack so the reduction function knows what kind of reduction to perform. The other queue receives the result: 21]

W. H. Dailey
47436 Mantes Street
Fremont, CA 94538

FORTH IN THE PUBLIC VIEW

After the survey article in March 15, 1979 Electronics, Mr. Robert Gaebler wrote the usual letter to the editor critiquing FORTH's postfix notation. We are reprinting a well stated rebuttal to this letter which also appeared in Electronics.

To the Editor:

I want to reply to Robert Gaebler's letter on expression format in the FORTH language [Electronics, July 5, 1979, p. 6].

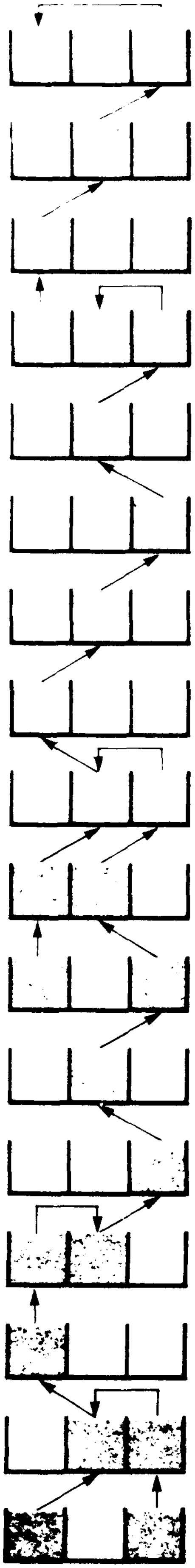
Gaebler notes, and I agree, that compilers can do the translation from infix to postfix notation and thus save the programmer both work and the risk of errors. Unfortunately, these advantages are not available without some penalty for extensible languages such as FORTH. If the compiler is to translate, it must know how to parse expressions. The parsing rules for primitive operators are supplied with the compiler, but those for the added operators must be supplied by the programmer at compile time, which makes the parser much more complicated.

Examination of almost any program will reveal that the majority of program statements are nonalgebraic or can easily be converted to a non-algebraic form. Thus the advantages of infix notation, when present, apply only to a fraction of the program statements. For most function definitions, the prefix notation of subroutine or macro calls is required, and this can be replaced by postfix notation with little or no loss of clarity.

Use of postfix notation leaves the parsing of all expressions in the hands of the programmer. It means that arguments for an operator may be prepared using the full power of the programming language, without any restrictions being imposed by the compiler. With this freedom comes the possibility of error, and argument preparation is one of the most error-prone portions of programming in a language such as FORTH. If effort is to be spent on improving the ease of programming, it should be spent on simplifying argument preparation and stack manipulation. Postfix notation, with the applicative style of programming that it produces, has so many advantages that it should not be sacrificed to an algebraic notation that is not "natural," but only something we all learned in school.

THIS IS THE END!
THE END OF VOLUME II #1!
THE END OF YOUR MEMBERSHIP?
DON'T LET IT HAPPEN!
RENEW TODAY!
CHECK THE LABEL FOR RENEWAL DATE!
SEND A CHECK TO FIG TODAY!
MAKE THIS YOUR BEGINNING!
RENEW NOW!

MAKE A COPY FOR A FRIEND!
POST COPY ON YOUR BULLETIN BOARD!



FORTH DIMENSIONS

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

Volume II
Number 2
Price $2.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No. 2 July/August 1980

Publisher Roy C. Martens

Editorial Review Board

Bill Ragsdale
Dave Boulton
Kim Harris
John James
George Maverick

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL 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 $12.00 per year ($15.00 overseas). 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 for 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, although our membership of 1100 is world-wide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

PUBLISHER'S COLUMN

Summer is here. Lazy days should abound but not at FIG and FORTH DIMENSIONS. Individuals and groups are working hard on Standards and Cases. Soon there will be announcements and printing of these efforts. In fact, if everything works out right, the next issue of FORTH DIMENSIONS will be the big one that everyone has been waiting for (have you renewed your subscription and membership?). This doesn't mean that we aren't eager for more articles and letters from members, send them. More issues are in the works.

Now that we have a few issues of the new looking FORTH DIMENSIONS under our belt, we'd like to have your suggestions about improvements and additional information. Do you want more technical material? More beginning input? More new product information? More????? Let us know. Remember your inputs are what make FORTH DIMENSIONS.

Have a nice summer. Renew if you haven't already.

Roy Martens

LYONS' DEN

While listening to the tapes of the FORTH Convention (available from Audio Village, P.O. Box 291, Bloomington, IA 47402, $16.00 for four tapes) I noticed puzzlement over how to communicate concisely the nature of FORTH, that is, what single term—operating system, compiler, interpreter—indentifies the class to which it belongs. How about referring to FORTH as a Meta Interpreter—a program for generating an interpreter (the application) to provide an interactive tool for solving application specific problems (sometimes referred to as JOL's, job-oriented-languages)? Other members of this class are LISP and an obscure IBM system called PLAN, as well as APL. FORTH has unique features distinguishing it from other members of this class, being more optimized for arithmetic than LISP, for example, and being more compact and lower level than APL. Also, its implementation is more like LISP than APL.

Continued on pg. 31

TEMPORAL ASPECTS OF THE FORTH LANGUAGE

A BEGINNER'S STUMBLING BLOCK

John M. Derick
Linda A. Baker
P.O. Box 553
Mountain View, CA 94042

Novice FORTH programmers who have had previous experience with other, more traditional, programming languages almost invariably become confused when first dealing with FORTH. A first time user sitting down at a FORTH terminal soon notices what seem to be time-based inconsistencies. That is, the language seems to require that things be done in the wrong order or that the language itself does things out of time order. The novice, striving to understand these supposed "inconsistencies" detects time as a note of commonality and therefore lumps them all together as one oddity, while in actuality there are three separate areas of difficulty.

The interesting point of this is that the cause of this confusion is so elementary that once the problems are understood, it is difficult to look back and pinpoint why the confusion arose in the first place. This is why these elementary problem areas are not stressed in most existing FORTH literature and are just assumed to be part of the longer than normal learning curve associated with FORTH. Making it clear in the neophyte's mind that there are three separate, but related, factor shortens this learning curve.

Let us examine what situations cause this confusion.

Sitting at a FORTH terminal, you enter a FORTH word, hit a carriage return and the word executes. Other times, though, you enter a line of FORTH words (including the one you just executed previously), hit carriage return and nothing executes. But when used later on this same word executes! As you learn more, you discover that in order to perform some functions you must actually alter the traditional time sequence of programming and modify FORTH's compiler after it already works and is ebugged. Then, to add even more confusion, you find that some words, when added to the compiler, will execute different parts of that same word at different times. Or, when you edit a FORTH program, save it on disk and then compile it; some parts compile as expected but other words execute immediately.

To an experienced FORTH programmer it is quite obvious that there are actually three separate (but releated) aspects of FORTH represented in this example. To a beginner all of these attributes are lumped together in one tangled question of "who's on first????" and "when did he get there????"

With the exception of different parts of a word executing at different times, these are very trivial problems to an experienced FORTH programmer. To the beginner they are totally new concepts that must be sorted out and grasped--even though once understood they really are trivial concepts.

Let us first address the most basic of these three time related stumbling blocks; that of modifying the compiler.

Before we continue it is important to point out that there are several steps (one may almost say laws) that always must be followed to generate object code from source code. Traditional programming languages take these steps in a straight line one-pass manner. FORTH also takes these same steps (i.e., a compiler has been written and installed). The difference with FORTH however, is that the act of writing the compiler is not intended to be a one-pass step. Instead it is a recursive procedure where the compiler is constantly modified and tailored to the users needs over and over again. This alters the time sequence of things and is a slightly shocking concept but the basic rules are still the same.

In traditional languages a programmer goes through several temporally separated steps to generate a user program: A compiler (or assembler), an editor, a link editor and loader are all separately created and installed on the user's system. Then the user edits a program, compiles it, links it, then loads and tests it. Everything is done in such absolutely clear cut steps that one is subtly led to believe that this is the absolute nature of the world.

FORTH on the other hand is a highly interactive, dictionary-based language where new additions to the language (i.e., user added words) are simply added to the end of the dictionary thereby "extending" it. FORTH's compiler is part of this dictionary and therefore words added to the dictionary can actually affect or be used in the compiler. In FORTH, this is not only possible, it is required if one is to fully use the power of the language.

A simple concept? Yes. But it is so contrary to traditional practice that it is hard for a neophyte to believe advanced documentation which tells how to build compiler directives such as "creating" or "defining" words while only alluding to the fact that the compiler can and should be modified.

Therefore, let us emphasize this fact: The compiler in FORTH is not sacred. The traditional sequential steps of writing a compiler and forever using that particular product do not apply in FORTH. FORTH's compiler may be modified at any time. All, or part of it may be executed at any time. As a matter of fact "creating" or "defining" words used in the compiler are actually tiny standalone compilers in themselves and can be used to perform mini-compilations whenever they are referenced.

Now that this compiler modification aspect has been "factored" out of the jumble of time related "confusions", the beginner is still left with the second point of confusion: Namely why words sometimes execute immediately and sometimes do not.

The technical reason why words execute immediately is that the "precedence" bit associated with that word is set on; but it is the philosophical reasoning for the existence of the precedence bit that is of importance to the neophyte.

Again all of this is tied in with the fact that FORTH's compiler is an integral interactive part of the language. It is an integral part of the language because it is composed of common FORTH words used not only in the compiler but in every other FORTH application as well.

Entering a FORTH word or words on a terminal, and hitting carriage return causes that word or words to be immediately executed and is similar to executing an already compiled and linked object module. The dictionary is searching until the word is found and the definition is executed. To do this, the word is preceded by a colon, FORTH is put into the compiler state, and all words up until a semicolon will be compiled (i.e., placed into the dictionary for future execution). This is similar to inputting source code to a FORTRAN compiler and getting object code out.

The point being made here is that FORTH continuously changes between "compiler" state and "execution" state. When in compiler state, most input words are compiled, not executed. Notice the word "most". Some words are executed while in compiler state. These are naturally called compiler words.

These compiler words are identical in appearance to any other FORTH word. Indeed they actually are simply FORTH words with the exception that their precedence bit is set. They are analogous to assembly language pseudo ops or compiler directives. A pseudo op (like ORG) in assembly language gives direction to or is "executed" by the assembler; not the object code. It is never executed by the user program.

Thus, words in FORTH may be "flagged" to operate as pseudo ops. That is, they may be chosen to execute immediately and thereby perform some act of compilation upon other words in the definition; (even if they are imbedded inside of a string of source code--just as a pseudo op would do in assembly language). This "flag" is the precedence bit. When the FORTH interpreter detects that this bit is set, it will cause it associated word to be executed immediately, even while in compiler mode. Using the word IMMEDIATE just after a definition is the method used to set the precedence bit.

This is a very powerful feature of the FORTH language. It allows definitions to execute while in compile mode and since FORTH makes no distinction between "supplied" words and user written words the compiler itself can be added to and improved. This feature is called "extendability".

There are certain defining words in FORTH that take the trait of "when a word is executed" one step further. Conceptionally advanced word such as BUILDS and DOES allow a definition to be constructed so that the first half of the word will be used at compile time but the second half will execute at execution time.

While it is beyond the scope of this paper to go into the usage of BUILDS and DOES type words, it should be noted that they exist and really do have two separate times of execution.

The last point of confusion is: When words contained in a "loaded" block execute immediately instead of compiling (or visa versa). When FORTH loads a block, it treats the incoming data almost as if it were being read from a keyboard. Definitions are compiled and put into the dictionary as they are encountered in the data stream. But, if a word is encountered that is not contained inside of a definition (whether intentionally or not!) that word is executed immediately, just as if it was entered from the keyboard. This is a quite straight forward, and quite understandable effect once it is pointed out. The rule here is to put words to be compiled inside of definitions. Leave words to be executed immediately outside of definitions.

A good example of word purposely left outside of a definition is DECIMAL. This word is normally used as the last word of a loaded block to insure that after compilation the system is left in its standard base ten state.

In summary, the temporal confusion that occurs when first using FORTH is all quite elementary and understandable--at least in principle. And at a beginners stage, principle is very important.

The three general categories; modifying the compiler, compiler directives using the precedence bit, and loading and compiling blocks, all perform execution at predictable times and really do have a direct correspondence with traditional programming sequences.

A GENERALIZED LOOP CONSTRUCT FOR FORTH

For some time, I have been building my own version of a FORTH-like language with direct rather than indirect threaded code, running on the 8080. Last year I learned that my approach is almost identical to that of URTH; this is not surprising since the design criterion of highest possible execution speed was the same. To this end, the inner interpreter has one level of indirection removed (compared to FORTH) and jumps (as for IF , ELSE , LOOP , WHILE , UNTIL , etc.) are compiled to their 16 bit absolute value, rather than a 16 bit offset. All this by way of preface that although my "home base" is isolated from the West Coast and my implementation of the following words may not be exactly FORTH compatible, yet I feel that the concepts presented are new and useful in the FORTH environment.

The article 'FORTH-85 "CASE" STATEMENT' by Richard B. Main in FORTH DIMENSIONS, Volume 1, Number 5 had a catalytic effect in the development of these ideas, specifically the technique of saving an unknown number of addresses on the stack and using zero as a marker for the last address. It seemed to me that one area to apply this scheme with good effect is in the BEGIN ... UNTIL and BEGIN ... WHILE ... REPEAT loop constructs which currently permit only one exit test. This sometimes forces awkward stack manipulations to "or" conditions when two or more conditions must be tested, any one of which is sufficient to terminate the loop. The proposed constructs solve this problem, require no more lower level CODE words than already exist, and add to the elegance of the language by removing the word REPEAT.

The generalized loop is constructed one of two ways:

```
        BEGIN ... WHILE ... WHILE ... WHILE ... UNTIL
  or
        BEGIN ... WHILE ... WHILE ... WHILE ... AGAIN
```

There can be any number of WHILE words in each loop, including none. The meaning of the words BEGIN , WHILE , UNTIL , and AGAIN is exactly the same as currently understood; no new concepts need be learned. For newcomers to the language (of which we all hope for, and in large numbers) the learning task is easier because we have reduced the number of FORTH basic words while at the same time increasing the power of the language by permitting more powerful combinations of these words. This is surely a good direction since the human (programmer) mind is unsurpassed at manipulating symbols, but not in remembering them.

The Words

The following definitions work in my system. In FORTH, where XELSE and XIF require a compiled offset rather than an absolute address, the words WHILE , COMPADDS , AGAIN , and UNTIL must be changed slightly.

```
( GENERALIZED LOOP WORDS - BEGIN WHILE UNTIL AGAIN )

: BEGIN     HERE 0                       ; IMMEDIATE
: WHILE     LIT XIF , HERE 0             ; IMMEDIATE
: UNTIL     DROP LIT XIF , ,             ; IMMEDIATE ( TEMPORARY
: COMPADDS  BEING DUP IF HERE 1+ 1+ SWAP ! ENDIF 0= UNTIL ;
: AGAIN     LIT XELSE , COMPADDS ,       ; IMMEDIATE
: UNTIL     LIT XIF , COMPADDS ,         ; IMMEDIATE
```

How They Work, Compile Time

BEGIN Pushes onto the stack the address to which the loop should jump, followed by a zero. The zero is used as a market by the COMPADDS word.

WHILE (if used) Compiles a conditional jump to the temporary address of zero, and also pushes the address of the temporary address to the

	stack. The temporary address, which can never be zero, will later be overwritten by COMPADDS with the address of the next word immediately after the loop structure; this is how WHILE effects a loop exit.
UNTIL	(temporary) Allows correct compilation of the COMPADDS word's BEGIN ... UNTIL structure. It will shortly be replaced with the generalized UNTIL .
COMPADDS	Overwrites the address of all previous WHILE words until the last BEGIN . Each address on the stack (there may be none) is overwritten with the vale HERE+2. The zero placed on the stack by the last BEGIN terminates the overwriting and leaves the address of the first word in the loop on the top of the stack.
AGAIN	Compiles an unconditional jump, completes all previous WHILE words, and then compiles the address of the unconditional jump, pointing to the top of the loop.
UNTIL	Identical to AGAIN , except a conditional jump is compiled, allowing a conditional loop exit.

How They Work, Run Time

They work the same as the previously known BEGIN , WHILE , UNTIL , and AGAIN .

Error Procedures

Error checks can easily be added to these words. This is done as below:

```
( GENERALIZED LOOP WORDS - BEGIN WHILE UNTIL AGAIN )
( WITH ERROR PROCEDURES AS PER RULL-HOLLAND )
: BEGIN      HERE 0 1                              ; IMMEDIATE
: WHILE      1 ?PAIRS LIT XIF, HERE 0 , 1          ; IMMEDIATE
: UNTIL      DROP DROP LIT XIF , ,                 ; IMMEDIATE ( TEMPORARY )
: COMPADDS   BEGIN DUP IF HERE 1+ 1+ SWAP ! ENDIF 0 UNTIL ;
: AGAIN      1 ?PAIRS LIT XELSE , COMPADDS ,       ; IMMEDIATE
: UNTIL      1 ?PAIRS LIT XIF , COMPADDS ,         ; IMMEDIATE
```

The operation is self-evident.

Conclusion

Generalized loop words BEGIN , WHILE , UNTIL , and AGAIN have been proposed. Their use provides, as a subset, the well known actions of BEGIN ... AGAIN , BEGIN ... UNTIL , and BEGIN ... WHILE ... REPEAT (with the word REPEAT replaced by AGAIN). When used in this manner the new words impose no more run time overhead in time or space than the words they replace. If the new words did nothing more, they would still be desirable because they "orthogonalize" the unconditional loop termination word, making it AGAIN regardless of the presence or absence of the WHILE word.

But, as an added benefit of the new words, more powerful constructs such as BEGIN ... WHILE ... UNTIL or BEGIN ... WHILE ... WHILE ... AGAIN are possible. Thus multiple tests and exits from a loop can be arranged in the most natural order, without the need to "or" the results of the tests. These multiple loop exits do not violate the principles of structured programming since they all lead to a common point; in other words, the loop, as a structure, has one entry and one exit.

Future Research

After much thought about the implications of the proposed words in relation to the FORTH philosophy of programming, I must say that of the two changes wrought by these words, viz.

and orthogonalization of the loop construct, and the ability to have multiple loop exits, I believe that orthogonalization is by far the most important result. In FORTH, while the very act of programming consists of extending the language by creating many new words useful in the application environment, even so, I believe that the initial basic words, especially the structured programming constructs such as IF ... ELSE ... ENDIF , BEGIN ... UNTIL , and DO ... LOOP should be as few and as general purpose as possible.

In addition, they should be carefully names so as to convey their action to programmers new to FORTH, but familiar with similar structures on other, "industry standard" languages such as ALGOL, PASCAL, and C. The construct IF ... ELSE ... THEN is poor in this respect; the word THEN confuses novices to FORTH since it usually implies selection, while in this case it is really a construct terminator. I assume that this is the reason why the change from THEN to ENIF was specified in FORTH-79. Similarly, BEGIN ... END is confusing since it does not imply repetition to the average programmer. FORTH-79 partially corrects this confusion with BEGIN .. UNTIL , but I believe some word signifying repetition should replace BEGIN , such as REPEAT ... UNTIL , REPEAT ... AGAIN , and REPEAT ... WHILE ... AGAIN .

As for DO ... LOOP , this construct cries out for a convenient way to prematurely exit the loop. LEAVE seems weird - at odds with commonly accepted practice - since it has a deferred effect, taking place only at the end of the loop. Although I won't remove it from the language, I suggest an alternative: Do ... WHILE ... LOOP . At the execution of the optional WHILE , if the stack is zero the loop is exitted. Not possible because WHILE is already used for the REPEAT ... WHILE ... AGAIN loop, you say? But it is possible! A very useful by-product of the Error Procedures of University at Ulrecht, Netherlands is that they always leave at the top of the stack (during compile time) a flag indicating the identity of the innermost construct, different for REPEAT ... and DO ...; it is then a simple matter to arrange WHILE to have different actions and to compile entirely different CODE words depending on this value. Of course, we would not limit the number of WHILE words between DO and LOOP . LOOP must be modified, as was described above for AGAIN , to permit this.

Bruce Komusin
Ontel Corp.
250 Crossways Park Dr.
Woodbury, NY 11797

New Product

OmniForth, from Interactive Computer Systems, is now available for the North Star computer. FORTH combines structured programming, stack organization, virtual memory, compiler, assembler, and file system into an extensible macrolanguage. Organized as a dictionary of words, FORTH allows defining new words that extend the vocabulary to suit any application. Words are compiled on entry into code ready for immediate test, and execute ten times faster than Basic. FORTH supports coding time-critical routines in assembler for the fastest response. OmniForth contains the interactive FORTH compiler (modeled on Fig-FORTH), assembler for the 8080 and Z-80, file system, and text editor. Omni-Forth requires 24K memory and North Star DOS, and costs $49.95; an optional Introduction to FORTH manual is available for $15.00. Interactive Computer Systems, Inc., 6403 DiMarco Road, Tampa, FL 33614.

FILE NAMING SYSTEM

Peter H. Helmers
University of Rochester

This particular FORTH file naming system is set up to use a disk based directory to name files which are comprised of a series of disk blocks. The system does not include any specific file formats, but instead is used to translate a filename to a block number. This block number can be a traditional "load block", a directory block for a linked set of random data blocks, or perhaps the initial block in a multi-block text file. Routines are available to control a disk's bit map of allocated blocks so that already utilized blocks are not overwritten. Additional routines allow creation of filename/block entries at either fixed block locations or at random locations, or deletion of file entries, directory listings, etc.

The philosophy in writing this package was that file formats should be user definable although several standard uses are being brought up for text files, and data arrays stored in consecutive blocks. By using the words available, additional file formats can be easily added.

The file naming system presently uses three blocks at the end of each disk. The first block contains two data arrays: a bit map of block usage on the disk, and a list of block-pointers for each defined filename. The bitmap uses one bit per disk block to define whether the block is used or not; the bit is a "1" if the block is used. The block pointer array consists of 64 integers which point to the filename's starting block number. A value of -1 means that the filename is undefined.

The second two blocks contain 64 filename strings of up to 32 characters each. Each name string is actually stored as a fixed length 32 byte string with any extra characters being padded blanks. A non-valid file is flagged by a -1 value for the block pointer, not by a null of special string.

The following is a list of the primary user oriented words in this file naming package:

("STR") <u>FIND-NAME</u> (INDX)

FIND-NAME searches for the STR in the directory and returns its directory index if found, or a -1 if not found. Thus a user can test for a -1 to see if a filename exists.

<u>INIT-DIRECTORY</u>

INIT-DIRECTORY is used to set all block pointers to -1's so that no files will be considered to be in existence.

<u>INIT-BIT-MAP</u>

INIT-BIT-MAP is used to set all bit map bits to 0's, thus indicating that no disk blocks are being used.

(BLK#) <u>FREE-BLK</u>

FREE-BLK is used to reset a given block's bit map bit, thus indicating that it is not in use.

(BLK#) <u>RESERVE BLK</u>

RESERVE-BLK is used to set a given block's bit map to indicate that it is in use.

<u>FIND-FREE-BLK</u> (BLK#)

FIND-FREE-BLK is used to find the first free block encountered in the bit map. It returns a "free" block number if one can be found, or a -1 if the disk is full.

("NAME") NEW

NEW is used to create a new filename entry with a block pointer found from the first free block encountered in the bit map.

("NAME"), (BLK#) NEW FIXED

NEW-FIXED is used to define a new filename with a specific block pointer (for example, a traditional "load block").

("NAME") FILE (BLK#)

FILE is used to translate a filename string to a specific block number.

("NAME") ERASE

ERASE is used to erase the given filename from the directory.

DIRECTORY

DIRECTORY is used to print a listing on the console of all defined filenames.

```
(FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: FILE-ERROR
   DOCASE
            DUP 1 = WHEN T" ALL BLOCKS USED "
      CASE DUP 2 = WHEN T" FILE ALREADY EXISTS "
      CASE DUP 3 = WHEN T" DIRECTORY FULL "
      CASE DUP 4 = WHEN T" NAME TOO LONG "
      CASE DUP 5 = WHEN T" FILE NOT FOUND "
   ENDCASE
   CR
   RESTART
;
2DROP ( DO CASE BUG )
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 4 79 ) BASE @ HEX
: PB O DO T"  " LOOP ;
: "SPACES O DO "  " "+ LOOP ; ( ADD [TOS] SPACES TO STRNG )
: "GET 20 SWAP "@F ;  ( GET 32 BYTE STRNG FROM ADDR ON TOS )
: "PUT 20 SWAP "!F ; ( PUT 32 BYTE STRNG TO ADDR ON TOS )
: FILE-NAME-FIX               ( MAKE NAME 32 CHARS LONG )
   "LEN SUP 1E >              ( CHECK THAT NAME <= 30 CHARS )
   IF
      4 FILE-ERROR            ( NPOE, SO GIVE ERROR )
   THEN
   20 -- "SPACES              ( PAD W/BLNKS TO 32 CHARS )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 30 NOV 79 ) BASE @ HEX
OF8 CONSTANT DIR ( FILE DIRECTORY BLOCKS START HERE )
: INDX->STR-ADDR            ( INDX ON TOS ON ENTRY )
   20 /MOD                  ( 32 FILENAMES/BLOCK )
   DIR + 1+                 ( NAMES IN BLKS DIR+1,DIR+2 )
   BLOCK                    ( ADDR OF BLOCK W/ NAME IN IT )
   SWAP 5 <-L               ( BYTE OFFSET INTO BLOCK
   +                        ( RTRN ADDR OF NAME STRING ON TOS )
;
: INDX->BLK-PTR-ADDR        .( INDX ON TOS )
   1 <-L                    ( CREATE BYTE OFFSET INTO BLOCK )
   DIR BLOCK                ( ADDR OF BLOCK WITH FILE POINTERS )
   +                        ( RTRN ADDR OF FILE'S BLOCK PNTR )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 11 30 79 ) BASE @ HEX
-1 VARIABLE FILE-INDX   -1 VARIABLE FILE-BLK
: FIND-NAME -1 FILE-INDX !  ( SET INDX FOR NO MATCH )
   40 0 DO                  ( CHECK ALL POSSIBLE NAMES )
      I INDX->BLK-PTR-ADDR @ -1 =
      IF                    ( VALID FILE - SO CHECK NAME MATCH )
         "DUP I INDX->STR-ADDR "GET "=
         IF                       ( NAME MATCH FOUND )
            I FILE-INDX ! EXIT ( SET INDX AND ESCAPE )
         THEN                     ( OTHERWISE )
      THEN
   LOOP                        ( TRY NEXT NAME ENTRY IF NOT DONE )
   "DROP FILE-INDX @           ( REMOVE TARGET STRING AND ... )
                               ( RETURN THE INDX OF THE STRING )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 11 30 79 ) BASE @ HEX
: CREATE-NAME               ( FILE-NAME STRING ON TOS )
   -1 FILE-INDX !           ( SET TO INDICATE NO ROOM AVAIL )
   40 0 DO                  ( SEARCH DIRECTORY FOR NULL FILE )
      I INDX->BLK-PTR-ADDR @ -1 =
      IF                    ( NULL, SO PLACE NAME HERE )
         I FILE-INDX !      ( SAVE INDX WHERE NAME IS SAVED )
         I INDX->STR-ADDR "PUT  ( SAVE FILE'S NAME IN DIR )
         "" UPDATE EXIT     ( NULL STR TO TOSS, AND EXIT )
      THEN
   LOOP                     ( UNTIL MATCH OR END OF DIR )
   "DROP FILE-INDX @        ( DROP TARGET OR NULL STRING, & )
                            ( RTRN INDX OF NEW FILE )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 11 30 79 ) BASE @ HEX
: DELETE-FILE               ( DELETE FILE GIVEN BY INDX ON TOS )
   INDX->BLK-PTR-ADDR       ( FIND ADDR OF BLK'S POINTER )
   -1 SWAP !                ( FLAG DELETION BY -1 BLK PTR )
   UPDATE                   ( FORCE DISK UPDATE )
;
: INIT-DIRECTORY            ( -DELETE ALL DIR ENTRIES )
   40 0 DO                  ( INDX ALL 64 DIR ENTRIES )
      I DELETE-FILE         ( DELETE EACH BY INDX )
   LOOP
;
BASE !   ;S
```

A Riddle

FORTH Supervisor: What's the difference between 'ignorance' and 'indifference'?

FORTH Programmer: I don't know and I don't care.

```
( FILE NAMING SYSTEM - PHH - 11 30 79 ) BASE @ HEX
: GET-BIT-MASK                   ( GET BIT MAP INFO FOR BLK# ON TOS )
   DUP
   7 & 1 SWAP <-L                ( GENERATE BIT#, THEN BIT MASK )
   SWAP 3 ->L                    ( GEN. BYTE OFFSET IN BIT MAP )
   300 +
   DIR BLOCK +                   ( ADD BIT MAP OFFSET W/IN DIR BLK )
   DUP C@                        ( DUP IT, AND GET ITS VALUE )
   ROT                           ( RTRN BIT MAP ADDR, OLD BIT MAP )
;                                ( BYTE, & BIT MASK ON TOS )
: FREE-BLK                       ( BLK# ON TOS TO BE FREE'D )
   GET-BIT-MASK
   -1 XOR & SWAP                 ( MASK BLK'S BIT MAP BIT TO 0 )
   C! UPDATE                     ( STORE BACK IN BIT MAP & TO DISK )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: RESERVE-BLK                    ( MARK BLK ON TOS AS USED )
   GET-BIT-MASK
   OR SWAP C! UPDATE             ( SET BIT IN BIT MASK )
;
: INIT-BIT-MAP                   ( FREE ALL BLKS, THEN RESERVE )
                                 ( THE RANGE OF BLKS GIVEN ON TOS )
   DIR 0 DO                      ( FREE ALL BLKS IN DISK )
      I FREE-BLK
   LOOP
   SWAP 1+ SWAP DO               ( RANGE OF BLKS ON TOS )
      I RESERVE-BLK              ( RESERVE ALL BLKS IN THE RANGE )
   LOOP
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 5 79 ) BASE @ HEX
: FIND-FREE-BLK                  ( SEARCH BIT MAP FOR FREE BLOCK )
  -1 FILE BLK !                  ( FLAG RESULT FOR NO BLKS FOUND )
   DIR O DO                      ( NOW SEARCH ENTIRE BIT MAP )
      I GET-BIT-MASK & 0=        ( IS BLK IN USE? )
      IF                         ( NO, ... )
         I FILE-BLK ! EXIT       ( SO SAVE BLK#, AND EXIT LOOP )
      THEN
      DROP                       ( BIT MAP ADDR )
  LOOP                           ( TRY THE NEXT BLOCK )
  FILE-BLK @                     ( DONE, SO RETURN THE FOUND BLK )
;                                ( NOTE, -1 => NO BLKS FREE )
BASE !  ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: NEW                            ( SET UP NEW FILE W/ NAME ON TOSS )
   FILE-NAME-FIX                 ( FIRST, FORCE VALID NAME LEN )
   FIND-FREE-BLK DUP -1 =        (. MORE ROOM ON DISK? )
   IF 1 FILE-ERROR THEN          ( NO, ALL BLKS RESERVED )
   "DUP FIND-NAME -1 =           ( NAME ALREADY USED? )
   IF 2 FILE-ERROR THEN          ( YES, GIVE ERROR MESSAGE )
   CREATE-NAME DUP -1 =          ( PUT NAME IN DIR, IF NOT FULL )
   IF 3 FILE-ERROR THEN          ( DIR FULL ERROR )
   SWAP DUP RESERVE-BLK          ( SET NEW BLK, FOUND BY )
                                 ( FIND-FREE-BLK, AS RESERVED )
   SWAP INDX->BLK-PTR-ADDR ! ( STORE FILE'S BLK POINTER )
   UPDATE                        ( GO TELL IT TO THE DISK, TOO ! )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: NEW-FIXED                      ( LIKE 'NEW' EXCEPT BLK POINTER )
                                 ( GIVEN BY # ON TOS )
   FILE-NAME-FIX                 ( FORCE 32 CHAR LENGTH )
   "DUP FIND-NAME -1 =           ( NAME ALREADY EXIST? )
   IF 2 FILE-ERROR THEN          ( YES, SO GIVE ERROR MESSAGE )
   CREATE-NAME DUP -1 =          ( PUT NAME IN DIR, IF DIR NOT FULL )
   IF 3 FILE-ERROR THEN          ( DIR FULL, SO GIVE ERROR )
   SWAP DUP RESERVE-BLK          ( RESERVE BLK, GIVEN BY # ON TOS )
                                 ( ON ENTRY TO 'NEW-FIXED' )
   SWAP INDX->BLK-PTR-ADDR ! ( AND STORE BLK# AS FILE'S PTR )
   UPDATE                        ( GO TELL IT TO THE DISK ! )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: FILE                           ( XLATE FILE NAME ON TOSS TO BLK# )
   FILE-NAME-FIX                 ( FORCE 32 CHAR STRNG LEN )
   FIND-NAME DUP -1 =            ( FIND NAME'S DIR INDX )
   IF 5 FILE-ERROR THEN          ( NAME NOT FOUND IN DIR )
   INDX->BLK<PTR-ADDR @          ( GET NAME'S BLOCK # )
;                                ( AND RETURN ON TOS )
: ERASE                          ( ERASE NAME ON TOSS FROM DIR )
   FILE-NAME-FIX                 ( FORCE 32 CHAR STRING LENGTH )
   FIND-NAME DUP -1 =            ( GET NAME'S DIR INDX, IF ANY )
   IF 5 FILE-ERROR THEN          ( NAME NOT FOUND IN DIR )
   DUP DELETE-FILE               ( DELETE FILE GIVEN BY INDX# )
   INDX->BLK-PTR-ADDR @          ( GET THE OLD BLK POINTER )
   FREE-BLK                      ( ...AND FREE IT IN THE BIT MAP )
;
BASE !   ;S
```

```
( FILE NAMING SYSTEM - PHH - 12 3 79 ) BASE @ HEX
: DIRECTORY                      ( PRINT ENTIRE DIRECTORY )
   40 0 DO                       ( CHECK EACH DIR ENTRY )
      I INDX->BLK-PTR-ADDR @ (GET BLK PNTR )
      DUP -1 =                   ( IS IT AN EXISTANT FILE? )
      IF                         ( YES, SO PRINT ITS CONTENTS )
         I INDX->STR-ADDR        ( FIRST, GET THE ADDR OF THE NAME )
         "GET ".                 ( PUT IT ON TOSS, AND PRINT IT )
         5 PB . CR               ( PRINT 5 BLNKS, AND THE BLK # )
      ELSE DROP                  ( BLK NUMBER )
      THEN
   LOOP                          ( CONTINUE FOR ALL POSSIBLE FILES )
;
BASE !   ;S
```

LYONS' DEN

(Continued from pg. 22)

Regarding FORTH this way captures some of the reasons why FORTH should not be used as merely a low level pseudo-machine in the way Wirth used P-Code to implement PASCAL, or as how meta compilers, as opposed to how a meta interpreter works. Of course, any language can be used to write an interpreter, but FORTH provides tools for this purpose built in and is thus pre-structured for that kind of application. This may also suggest—as just a possibility—why there has been observed markedly less use of conditional branches in FORTH programs relative to FORTRAN; perhaps many of the conditionals that would be explicit in FORTRAN are simply performed as executions of the interpreter functions which perform a complex set of conditional branches automatically without having to identify them as such. I will wager LISP is the same way.

George B. Lyons
Jersey City, NJ

TOWERS OF HANOI

by Peter Midnight

Here are the listings of a graphic representation of the ancient Towers of Hanoi puzzle which is adjustable for any CRT terminal with curser addressing.

Recently, when I got fig FORTH running on my system under North Star DOS, I decided to translate this program into FORTH as an exercise and as a comparison between FORTH and PASCAL. In the process I noticed some inefficiencies but chose to translate them more or less directly, for the sake of comparison.

The UCSP PASCAL program is available by requesting the Jan/Feb 1980 Newsletter from Homebrew Computer Club, P.O. Box 626, Mountain View, CA 94042.

Forth Program

```
SCR # 12
  0 ( TOWERS OF HANOI     Copyright, 1979, Peter Midnight )
  1 ( Translated for speed comparison ) FORTH DEFINITIONS DECIMAL
  2 ( First extend Forth to include a few features of Pascal )
  3 : MYSELF  ( In definition, this is a recursive use of new
  4      LATEST PFA CFA , ; IMMEDIATE                        word )
  5 : GOTOXY    ( X Y GOTOXY )    27 EMIT  61 EMIT
  6   0 MAX  15 MIN  32 +  EMIT   0 MAX  63 MIN  32 +  EMIT  ;
  7 : CLEARSCREEN  12 EMIT ;
  8 : 2DROP  DROP DROP ;
  9 : PICK  SP@  SWAP  2 *  +  @  ;
 10 : 4DUP  4 PICK  4 PICK  4 PICK  4 PICK ;
 11   10 CONSTANT NMAX  ( maximum permisable number of rings )
 12 NMAX VARIABLE (N)   : N (N) @ ; ( formerly a constant )
 13    0 CONSTANT HELL_FREEZES_OVER   43 CONSTANT COLOR ( + )
 14    0 VARIABLE RING   N 2 -  ALLOT  ( array [1..N] of bytes )
 15   -->

SCR # 13
  0 ( TOWERS OF HANOI     Copyright, 1979, Peter Midnight )
  1 : DELAY   ( centiseconds DELAY )
  2   0 DO  17 0 DO  127 127 * DROP  LOOP  LOOP  ;
  3 : POS     ( location POS -> coordinate )
  4   2 N *  1+  *  N +  ;
  5 : HALFDISPLAY   ( color size HALFDISPLAY )
  6   0 DO   DUP EMIT   LOOP   DROP  ;
  7 : <DISPLAY>   ( line color size <DISPLAY> )
  8   2DUP HALFDISPLAY   ROT 3 <  IF BL  ELSE 124 ( | )
  9   THEN EMIT   HALFDISPLAY  ;
 10 : DISPLAY   ( size pos line color DISPLAY )
 11   SWAP >R ROT ROT OVER - R ( color size pos-size line )
 12   GOTOXY   R> ( color size line )  ROT ROT <DISPLAY> ;
 13 -->
 14
 15

SCR # 14
  0 ( TOWERS OF HANOI     Copyright, 1979, Peter Midnight )
  1 : PRESENCE   ( tower ring PRESENCE -> boolean )
  2   RING +  C@  =  ;
  3 : LINE        ( tower LINE -> display_line_of_top )
  4   4 SWAP  N 0 DO  DUP I PRESENCE 0=  ROT + SWAP  LOOP DROP ;
  5 : 1-   1 - ;
  6
  7 : RAISE   ( size tower RAISE )
  8   DUP   POS  SWAP  LINE   1  SWAP  DO
  9   2DUP I BL DISPLAY   2DUP  I 1-  COLOR DISPLAY
 10   -1 +LOOP   2DROP  ;
 11 : LOWER   ( size tower LOWER )
 12   DUP   POS  SWAP  LINE 1+  2 DO
 13   2DUP  I 1-  BL DISPLAY   2DUP I COLOR DISPLAY
 14   LOOP  2DROP  ;
 15 -->

MSG # 15

SCR # 15
  0 ( TOWERS OF HANOI     Copyright, 1979, Peter Midnight )
  1 : MOVELEFT  ( size source_tower destiny_tower MOVELEFT )
  2   POS 1-  SWAP POS 1- DO  DUP  R 1+  1  BL  DISPLAY
  3   DUP R 1 COLOR DISPLAY   -1 +LOOP   DROP  ;
  4 : MOVERIGHT ( size source_tower destiny_tower MOVERIGHT )
  5   POS 1+  SWAP POS 1+ DO  DUP  R 1-  1  BL  DISPLAY
  6   DUP R 1 COLOR DISPLAY      LOOP      DROP  ;
  7 : TRAVERSE  ( size source_tower destiny_tower TRAVERSE )
  8   2DUP >    IF MOVELEFT  ELSE MOVERIGHT   THEN  ;
  9 : MOVE   ( size source_tower destiny_tower MOVE )
 10        ?TERMINAL IF  0 N 4 + GOTOXY  ABORT  THEN
 11   ROT ROT 2DUP RAISE   >R 2DUP R> ROT TRAVERSE
 12   2DUP  RING + 1-  C!    SWAP LOWER  ;
 13 -->
 14
 15

SCR # 16
  0 ( TOWERS OF HANOI     Copyright, 1979, Peter Midnight )
  1 : MULTIMOV  ( size source destiny spare MULTIMOV )
  2   4 PICK  1 =  IF DROP MOVE      ELSE
  3   >R >R SWAP 1- SWAP R> R>       4DUP SWAP MYSELF
  4   4DUP DROP   ROT 1+ ROT ROT   MOVE
  5   ROT ROT SWAP   MYSELF    THEN  ;
  6
  7 : MAKETOWER   ( tower MAKETOWER )
  8   POS  4 N +  3 DO  DUP I GOTOXY  124 EMIT ( | )  LOOP DROP ;
  9 : MAKEBASE    ( no arguments )
 10   0 N 4 + GOTOXY  N 6 * 3 +  0 DO  45 EMIT ( - )  LOOP ;
 11 : MAKERING    ( tower size MAKERING )
 12   2DUP RING + 1- C!   SWAP LOWER ;
 13 : SETUP   ( no arguments )   CLEARSCREEN
 14   N 1+ 0 DO  1 RING I + C!  LOOP   3 0 DO  I MAKETOWER  LOOP
 15   MAKEBASE   0 N DO  0 I MAKERING  -1 +LOOP  ;   -->

SCR # 17
  0 ( TOWERS OF HANOI     Copyright, 1979, Peter Midnight )
  1 : TOWERS   ( quantity TOWERS )
  2   1 MAX  NMAX MIN  (N) !
  3   SETUP   N 2 0 1    BEGIN
  4   OVER POS  N 4 +  GOTOXY   N 0 DO  7 EMIT  50 DELAY  LOOP
  5   ROT   4DUP   MULTIMOV
  6   HELL_FREEZES_OVER UNTIL  ;
  7
  8 ;S
  9
 10 ( Results:  DELAY runs much slower in Forth than in Pascal.
 11   But the rest of the program is over twice as fast in Forth!
 12
 13   Note that CLEARSCREEN and GOTOXY are terminal dependant.
 14   NMAX should be 10 for 16x64 or 12 for 24x80 screens. )
 15

MSG # 15
```

Thanks to "THE I/O PORT", the Official Newsletter of the Tulsa Computer Society, for the feature article on FORTH by Art Sorski in their April 1980 issue. Address: The Tulsa Computer Society, P.O. Box 1133, Tulsa, OK 74101.

LETTERS

I'd like to take this chance to accomplish several aims. First, let me congratulate Roy Martens and the entire editorial staff for a fine publication in FORTH DIMENSIONS.

My interest in FORTH is far from passive; I have been using the University of Rochester's (my employer, by the way) URTH dialect for several years now. While at first I used it mainly at home for a private music synthesizer research project, I have more recently been applying it with success to several laboratories within the University's Medical Center. The applications have primarily been concerned with slow speed (10 to 100 samples per second) analog data acquisition and analysis - the latter involving the use of the AMD 9511 IC for number crunching (and it is fast ...!). These data acquisition systems have been described in an article which I just recently submitted to BYTE for publication (I hope).

While using FORTH in these applications, I have developed a set of goals for the elimination of some of the limitations of FORTH (there are some, you know ...). One of the major problems has been saving only three characters plus the length for identifiers; I have just recently implemented changes to adopt (in URTH) the FIG standard. Using primarily S-100 hardware, I am also now implementing a hardward debug facility for FORTH which allows easier program development. The design is very simple, but allows traps at instructions, memory references, and/or I/O references. I consider this method of debugging immeasurably more useful than just software trapping at each pass through NEXT.

Additional FORTH changes planned are the implementation of a random block text file system with variable record length and blanks compaction. I feel that this system will make it easy to write programs in a more readable format since this better formatted text will use less space than the current block oriented text editors. Thus there will be less of a temptation to use a short, cryptic coding style. My method of blanks compaction is to use the MSB of each text character to flag a compaction count byte. When listing a program in the editor, the compacted blanks can be re-expanded while they can be interpreted as blanks (due to changes in the WORD routine in URTH) when loading the text. Text will be stored on disk blocks as an integral number of lines of text per block with each line being defined as 0 or more characters followed by a carriage return character.

Text will be able to span multiple random blocks to avoid any "artificial" program length constraints due to fixed block size. Blocks are associated together via a doubly linked (forward and backward) pointer scheme while block usage is kept track of via a bit map (more on this later) corresponding to the disk's block utilization. So far the text editor has been written, but not fully debugged. However, the bit map and filing name system has been written and used for several months. I'd like to discuss them here as the type of entity which should be standardized for FIG FORTH usage. Let me try to motivate this building of file structures by analogy to building data structures in FORTH.

IN FORTH (or at least URTH) one can use some system features to define any arbitrary data structure. One which I've used recently is:

```
   : IPARAM <BUILDS  2 ALLO7 DOES >
```

which might be used:

```
   IPARAM MY-VIRTUAL-INTEGER
```

The important things to notice in this example are that the IPARAM data type first uses standard dictionary features to add new specific variables - in this case MY-VIRTUAL-INTEGER - to the dictionary. IPARAM also sets aside some dictionary space - in this case just one word - to store data for MY-VIRTUAL-INTEGER. Thus there are two important actions here - that of linking a variable's name into the dictionary, and that of reserving dictionary space for a variable's storage requirements.

The file system that I have been evolving also achieves two analogous actions to those above. First, it has a way of linking a file's name into a diskettes name directory, and second, it has a way of reserving disk block space for a file's sole use. Note, that it does not concern itself in any manner with how the file is logically formatted. As such, it is not a complete file management system, but only a common protocol for various logical file structures!

Let me explore two uses of file types built on this foundation. The previously mentioned text file system logically builds a file structure by the use of doubly linked random blocks. But in another case, the file is logically built up as an array of consecutive integers in consecutive disk blocks - thus linked only implicitly. Other logical structures are as diverse as are FORTH data types.

In summary, what I am proposing to be discussed and hopefully standardized is a common structure which can be used to name files and reserve disk space for files. I am not suggesting any specific file structures or formats for standardization. I am enclosing a copy of the source listings and some (hastily written) documentation for this file system so that it might stimulate comments and improvements from the public domain.

Thanks very much, and keep up the good work....

Peter H. Helmers
University of Rochester
Rochester, N.Y.

In December I got tired of waiting and implemented FORTH-65 from the fig-FORTH model. By the end of December I had it up and running. This version follows the model exactly except for printer control, the disk kinkage, and the inner interpreter.

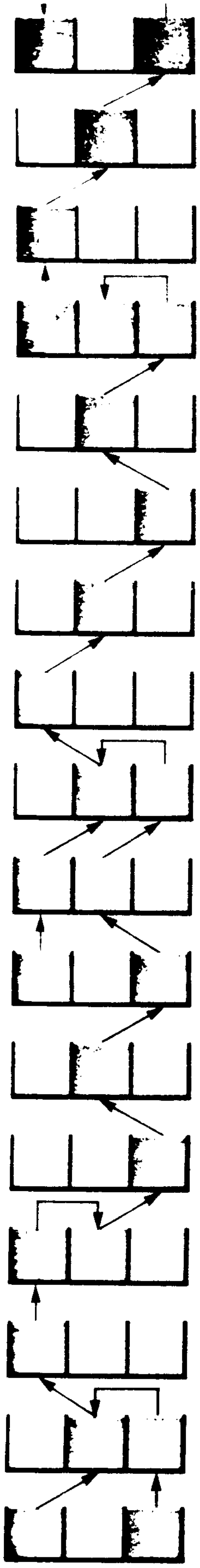
The jump indirect in the inner interpreter doesn't always work, JMP ($XXFF) doesn't work correctly on a 6502. If a CFA ends in $FF it's goodbye FORTH.

This bus bit after my third reassembly of FORTH-65. The inner interpreter I'm now using is considerably slower (60 cycles) but it is reliable.

I assembled FORTH-65 through the disk I/O (SCR #69), Screens 72 through 92 reside on disk and are compiled as needed. What I need now is the ASSEMBLER vocabulary. Has anyone done any work on a FORTH assembler for the 6502?

```
SCR #44
 ( RANDOM NUMBER GENERATOR E )

DECIMAL

0 VARIABLE SEED

: (RAND)   SEED @ 259 * 3 + 32767 AND DUP SEED ! ;

: RANDOM   (RAND) 32767 */ ;                (RANGE -1  )

;S
```

J.E. Rickenbacker
Houston, TX



FORTH DIMENSIONS

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

Volume II
Number 3
Price $5.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No. 3 September/October 1980

Publisher Roy C. Martens

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL 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 $12.00 per year ($15.00 overseas). 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 for 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,' although our membership of 1100 is worldwide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

PUBLISHER'S COLUMN

Busy, busy, busy. That's what its been for the last couple of months. Here are some of the things that have been happening.

1. We've reorganized the order processing for the mail order items listed on the last page. The volume has increased so much this year that we had gotten several months behind. Now, its all being handled at one location and we even have a phone number for checking on your orders (415) 962-8653. If you have technical questions DO NOT CALL, write to the box number so that your request can be routed to the most helpful person.
2. We can now take VISA and Master Charge orders by mail and by phone, (415) 962-8653. The charge on your monthly statement will be listed as "Mt. View Press". This was done because FIG isn't set up to handle charges. We still aren't ready to handle purchase orders or delayed billings.
3. The August issue of Byte magazine has put FORTH ON THE MAP. We are receiving 50-60 orders and requests for information a day. We have a supply of the issue and can furnish them to you (see the Mail Order form on the last page).
4. This Issue of FORTH DIMENSIONS has 60 pages and includes all the CASES that were submitted. Don't get your hopes up for more FD's this long. Next month we go back to our regular size. Congratulations to all entrants!
5. Two events are coming up soon. The FORML Advanced Conference will be held November 26-28, 1980 at Asilomar Conference Grounds, CA. The National FORTH Interest Group Convention will be held on November 29, 1980 at San Mateo, CA. See Page 94 for more information and to register.

My thanks to the Judges and Editorial Review Board for all the help they have given me on this BIG issue. Without their assistance, much of it done late at night, you wouldn't be reading this issue for months to come. Many thanks!

Roy Martens

CASE CONTEST CLOSES

This issue of FORTH DIMENSIONS is another special issue, chiefly devoted to FIG's CASE Statement Contest. The contest, announced in FD I/5, Jan./Feb. 1980, brought entries from sixteen individuals and teams, showing a high level of interest and activity among the membership.

All the entries are published here. They show imaginative thinking and hard work, and illustrate the many different ways that FORTH allows the user to implement a single concept. Although no one entry seemed to get it all together, many show some very good work.

Our panel of judges did not settle on a single winner, but instead have decided that the prize will be shared among three entries. These are Dr. Charles E. Eaker, Steve Munson, and the team of Karl Bochert and Dave Lion.

Each of these winners will receive a $50 prize and a one year subscription to Infoworld. The high interest in the contest has justified increasing the overall prize from the $100 announced (including $50 contributed by FORTH, Inc.) to $150. Infoworld kindly donated the subscriptions.

Eaker's entry is particularly well organized, and has a clear, readable writeup. He implemented a keyed CASE statement, and uses non-obvious words. (See below for the difference between positional and keyed CASE statements.)

Munson put so much thought into the contest that he included several versions, differing in the type of data that keys the CASE statement, and in keyed versus positional ordering of cases.

Bochert and Lion submitted a neat positional entry. It includes the ability to alter the binding of cases to case bodies after compile time.

The judging was based on a variety of factors:

1. the approach taken, including degree of generality;
2. the success and efficiency of the implementation, e.g., a minimum of computation and dictionary use should be left to execution time;
3. FORTH-like style, including good documentation on the screens;
4. overall prose description, together with an evaluation of the advantages and limitations of the approach or implementation;
5. adequacy and clarity of examples.

However, the judging did not involve loading and testing the entries on a running FORTH system.

The judges felt that most entrants were not getting close enough to what is possible in FORTH. They seemed to think along narrow lines. A general CASE implementation should be efficient both for the positional case (where the values tested are restricted to the first N integers, for example, similar to FORTRAN's computed GO-TO), and for the general "keyed" case, where a value, not necessarily an integer, is tested against a sequence of explicit values. Very few people tried to solve both.

This collection of contest entries make this issue of FD an excellent source for the comparative study of implementation techniques. Interested FORTH students should read each entry to pick up helpful techniques and evaluate style. (Caution: Any entry may also show poor techniques and weak style.)

Forth Dimensions welcomes more contributors.

JUST IN CASE

Dr. Charles E. Eaker

Even though FORTH provides a variety of program control structures, a CASE structure typically has not been one of them. There is no particular reason for this since, as we shall soon see, it is not difficult to implement one.

There are two different approaches one can take to implementing a CASE structure: vectored jumps and nested IF...ELSE...THEN structures. Vectored jumps provide the greatest speed at run-time but produce enormous compiling complications. So, taking the path of least resistance, here is a proposal for implementing a CASE structure for FORTH which is really just a substitute for nested IF structures. But, even though the proposal is logically redundant, there are a number of practical benefits which make it worthy of consideration.

To help this discussion, consider a word which might appear in an assembler vocabulary with a glossary entry as follows:

GEN operand, opcode, mode selector ---

Used by the ASSEMBLER vocabulary to generate opcodes. 'Mode selector' is the value which indicates which addressing mode has been specified. 'Opcode' is the value placed on the stack by the preceding mnemonic, and 'operand' is the value to be used as the argument of the opcode.

Here is one way of coding GEN.

```
: GEN  0 OVER =
   IF DROP IMMEDIATE
   ELSE 10 OVER =
    IF DROP DIRECT
    ELSE 20 OVER =
     IF DROP INDEXED
     ELSE 30 OVER =
      IF DROP EXTENDED
      ELSE DROP MODE-ERROR
      ENDIF
     ENDIF
    ENDIF
   ENDIF  RESET ;
```

GEN is defined to expect a 16-bit number on top of the stack. For each IF, this number, the "select value," is copied and tested against a constant, the "case value." If the select value equals the case value the appropriate code is executed. If all tests fail, MODE-ERROR is executed. Notice that GEN meticulously keeps the stack clean.

Depending on the select value, some action is performed on the opcode and operand, and GEN removes them from the stack. Consequently, before each test, GEN must copy (OVER) the select value, and if the test is successful, the select value must be dropped from the stack to expose the data values prior to the appropriate routine being called.

But wouldn't you rather code this thing this way?

```
: GEN CASE
          0 OF   IMMEDIATE    ENDOF
         10 OF   DIRECT       ENDOF
         20 OF   INDEXED      ENDOF
         30 OF   EXTENDED     ENDOF
         MODE-ERROR
      ENDCASE   RESET ;
```

It is certainly easier to see what this routine is doing, so comments are not as necessary, and changes and repairs are far easier to do. Here are the required colon definitions of CASE, OF, ENDOF, and ENDCASE.

```
CASE      ?COMP   CSP @  !CSP   4  ;  IMMEDIATE

OF  4 ?PAIRS  COMPILE OVER  COMPILE =  COMPILE 0BRANCH
       HERE 0 ,   COMPILE DROP   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
```

It so happens that with these definitions both versions of GEN compile the identical code into the dictionary. Let's look at the compiling details.

CASE makes sure that it is in a colon definition. Then it saves the value of CSP (which contains the position of the stack at the beginning of this case structure) and sets CSP equal to the present position of the stack. The new value of CSP will be used later by ENDCASE to resolve forward references. Finally, it throws a four onto the stack which will be used for checking syntax. CASE compiles no code into the dictionary.

OF first checks that it has been preceded either by CASE or an ENDOF. If the syntax is in order, then code is compiled into the dictionary to duplicate the select value (OVER) and test its equality to the current case value (=). Next, code for a conditional branch is compiled into the dictionary followed by code for DROP. Notice that at run-time the DROP is executed only if the select value equals the constant for this OF...ENDOF pair.

ENDOF first checks that an OF has gone before. If so, then it compiles an absolute branch to whatever code follows ENDCASE. However, the address to branch to is not yet known, so a dummy null is compiled into the address and its location is left on the stack so ENDCASE will know where to stick the address once it is known. But there is already an address on the stack just under the one which ENDOF just pushed. This address was left by OF and it points to an address that should hold a branch address to the code which follows the code generated by ENDOF. So, ENDOF swaps the addresses and calls ENDIF to resolve the address at the address left by OF. Finally, ENDOF leaves a four on the stack for syntax checking.

ENDCASE makes sure it has been preceded by either a CASE or ENDOF. Otherwise an error message is issued and compilation is aborted. Code for a DROP is compiled into the dictionary, then all the unresolved forward branches left by each ENDOF are resolved. Since there may be any number of them, including none, ENDCASE checks the current stack position against what it was when CASE was executed, and performs a fixup by calling ENDIF until the stack no longer contains addresses left by previous ENDOF's. Notice that all of these branches are resolved to point to the code after the DROP generated by ENDCASE. In the case of GEN this is RESET.

It doesn't take long to notice that OF generates an enormous amount of code (10 bytes). This is a classic example of a situation that cries out for a machine language primitive. If a run-time word could be defined, let's call it (OF), then each OF would generate just 4 bytes two to point to (OF) and two for the branch address. What (OF) would have to do is pull the top stack item (the current case value) and test it for equality with the new top stack item (the select value) If the test for equality is true then the next item on the stack the select value is also popped and execution continues after the (OF) If the test is false execution branches using the branch value following the pointer to (OF), and the select value is left on the stack.

```
CODE  (OF)  A PUL   B PUL   TSX
          1,X B SUB   0,X A SBC  ABA  0=
          IF  INS INS   ' BRANCH CFA 4 ( HEX ) 11 + JMP
          THEN   ' BRANCH CFA 2 JMP
: OF  4 ?PAIRS   COMPILE (OF) HERE 0 , 5 ; IMMEDIATE
```

The M6800 code listed above is straightforward except that is uses code in BRANCH and 0BRANCH. (OF) should work in any FIG 6800 installation provided BRANCH and 0BRANCH have not been altered (it doesn't matter where they are located). Non-6800 users will have to roll their own, but the high-level OF should make it clear what has to be done.

The disadvantages of this CASE proposal are that execution is not as fast as a vectored implementation, and in some versions of FORTH, ENDOF and ENDIF cannot be distinguished. These seem minor compared to the advantages - and there are several.

First, a CASE statement may contain any number of OF...ENDOF pairs, and the constants may be arranged in any order whatever. Actually the constants need not be constants. Between an ENDOF and the next OF the programmer may insert as much code as he or she likes including code which will compute the value of the "constant." CASE statements may be nested; a CASE...ENDCASE pair may appear between an OF...ENDOF pair. Furthermore, there need not be any code between CASE and ENDCASE, nor must there be code between OF and ENDOF. There must be code which pushes a 16-bit number to the stack prior to each OF. Finally, this proposal follows the fig-FORTH style of handling control structures.

fig-FORTH GLOSSARY

CASE --- addr n (compiling)

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At compile-time CASE saves the current value of CSP and resets it to the current position of the stack. This information is used by ENDCASE to resolve forward references left on the stack by any ENDOF's which precede it. n is left for subsequent error checking.

CASE has no run-time effects.

```
OF --- addr n        (compiling)
       n1 n2 --- n1  (if no match)
       n1 n2 ---     (if there is a match)
```

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, OF checks n1 and n2 for equality. If equal, n1 and n2 are both dropped from the stack, and execution continues to the next ENDOF. If not equal, only n2 is dropped, and execution jumps to whatever follows the next ENDOF.

At compile-time, OF emplaces (OF) and reserves space for an offset at addr. addr is used by ENDOF to resolve the offset. n is used for error checking.

ENDOF addr1 n1 --- addr2 n2 (compiling)

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, ENDOF transfers control to the code following the next ENDCASE provided there was a match at the last

OF. If there was not a match at the last OF, ENDOF is the location to which execution will branch.

At compile-time ENDOF emplaces BRANCH reserving a branch offset, leaves the address addr2 and n2 for error checking. ENDOF also resolves the pending forward branch from OF by calculating the offset from addrl to HERE and storing it at addrl.

```
ENDCASE addrl...addrn n --- (compiling)
          n ---      (if no match)
            ---      (if match was found)
```

Used in a colon definition in the form: CASE...OF...ENDOF...ENDCASE. Note that OF...ENDOF pairs may be repeated as necessary.

At run-time, ENDCASE drops the select value if it does not equal any case values. ENDCASE then serves as the destination of forward branches from all previous ENDOF's.

At compile-time, ENDCASE compiles a DROP then computes forward branch offsets until all addresses left by previous ENDOF's have been resolved. Finally, the value of CSP saved by CASE is restored. n is used for error checking.

```
(OF) nl n2 --- nl (if no match)
     nl n2 ---    (if there is a match)
```

The run-time procedure compiled by OF. See the description of the run-time behavior of OF.

Dr. Charles E. Eaker
Department of Philosophy
State University of New York
Oswego, NY 13126

Judges' Comments -

This is an excellent development and presentation of a key case statement with single integer keys. The following features make it immediately useful:

1. The reader can easily understand what the statement does and how to use it. There are only four words to learn, their functions are immediately clear from the example presented and their names are not confused with each other. (The ENDOF - ENDIF similarity will go away when the FIG model drops ENDIF in favor of the Standards Team decision to use THEN.)

2. One form of the statement can be entered entirely in higher-level fig-FORTH, and run immediately on any FIG system. An optional code word (for 6800) with redefinition of one of the four higher-level words saves run-time memory and time. Either way, the whole statement fits easily on one screen, including compile-time checking.

3. The narrative documentation is excellent. The glossary definitions are detailed (appropriate for this forum). For general distribution they could be condensed to user-only information.

This entry presents one kind of case statement out of several that are desired. We hope that this competent and straightforward work will serve as a model to future development.

COME TO FIG CONVENTION NOVEMBER 29

THE : CASE

Steve Munson

Having grown up on an ancient version of FORTH Inc. micro FORTH, I can appreciate the improvements rendered by fig-FORTH's renames and redefinitions. I was particularly impressed by the source equivalence of HERE NUMBER DROP which functions the same although in one case one is dropping the address of the first non-numeric delimiter, and in the other case one is dropping the most significant half of a double precision number!

My one beef is why was : made IMMEDIATE? Surely nobody wants a header in the middle of a colon definition. By the way, as you probably already know, this tends to mask an error in the definition of ; on the listing I have for the 6502 fig - FORTH. There is no [COMPILE] before the [which means compile mode is never terminated. In fact, I am not sure I see the point of the E property in your glossary. All words ought to be designed, at great pains if necessary, so that they can be compiled. My definition of CASE denies the E property of :, and I would be rash to assume no one would ever want to compile CASE.

Please find enclosed a listing, documentation, glossary entries, and a diskette. The diskette also contains the assembler used to generate the code, as it may be nonstandard If the fig-FORTH does not run on your system as it does on mine, feel free to edit my ideas into polished fig-FORTH (I am a novice figger) and re-list the screens; however I believe they will require no modification.

GLOSSARY ENTRIES

2BYTECASE

Keycase defining word, used in the form:

2BYTECASE cccc key$_0$ case$_0$ key$_1$ case$_1$ . . . key$_n$ case$_n$ (default case) END-CASE. Defines cccc as a caseword which expects a 2-byte key on the stack at run-time. If the key equals key$_0$ (a 2-byte key), case$_0$ (a previously defined word) will execute; if it matches key$_1$, case$_1$ will execute, and so on. The default case will execute on no match; if no default is specified, NOOP is assumed. Cases may be IMMEDIATE words, but no compile-time execution will occur within the case structure. The structure must be terminated by END-CASE. (See END-CASE, BYTECASE).

BYTECASE

Keycase defining word, used in the form:

BYTECASE cccc key$_0$ case$_0$ key$_1$ case$_1$ . . . key$_n$ case$_n$ END-CASE. Defines cccc as a caseword which expects a 1-byte key (most significant byte is ignored) on the stack at run-time. If the key equals key$_0$ (a 1-byte key), case$_0$ (a previously defined word) will execute; if it equals key$_1$, case$_1$ will execute, and so on. The default case will execute on no match; if no default is specified, NOOP is assumed. Cases may be IMMEDIATE words, but no compile-time execution will occur within the case structure. The structure must be terminated by END-CASE. (See END-CASE, CASE).

CASE

Case defining word, used in the form:

CASE cccc case$_0$ case$_1$ . . . case$_n$ (default case) END-CASE. Defines cccc as a caseword which expects a 1-byte key (most significant byte is ignored) on the stack at run-time. If the index is 0, case$_0$ (a previously defined word) will execute; if index is 1, case$_1$ executes, and so on. NOOP cases must be inserted for unused values of the index; index limit is 65, 535. No protection is made for out-of-range indices or stack underflow. CASE remains in compile mode (by calling :) until terminated by END-CASE. (See END-CASE).

DO-2BYTECASE P, C +

Compiles a reference to the run-time procedure of the same name, a two-byte "exit address", a one-byte case count and case structure identical to that of 2BYTECASE, all inline within a colon definition. Used in the form: : cccc optional words DO-2BYTECASE key$_0$ case$_0$ key$_1$ case$_1$ . . . key$_n$ case$_n$ (default case) END-CASE optional words ; . The keys, cases and run-time activity are exactly as described for 2BYTECASE. (See 2BYTECASE, END-CASE).

DO-BYTECASE P, C +

Copy glossary entry above, substituting BYTECASE for 2BYTECASE everywhere.

DO-CASE P, C +

Compiles a reference to the run-time procedure of the same name, a two-byte "exit address", and a case structure identical to that of CASE, all inline within a colon definition. Used in the form: : cccc optional words ; . The cases and run-time activity are exactly as described for CASE. (See CASE, END-CASE).

DO-STRINGCASE P, C +

Copy glossary entry for DO-2BYTECASE, substituting STRINGCASE for 2BYTECASE everywhere.

END-CASE P

Universal caseword delimiter. It has no run-time activity, but at compile-time it may fill in an "exit address" (inline caseword), and/or a case count (keycaseword), or terminate compile mode (CASE, inline CASE for CODE definitions).

STRINGCASE

Keycase defining word, used in the form:

STRINGCASE cccc key$_0$ case$_0$ key$_1$ case$_1$ . . . key$_n$ case$_n$ (default case) END-CASE. Defines cccc as a caseword which expects a byte-string beginning at HERE 1+ with a count of them at HERE (typically fetched by WORD) at run-time. If the string equals key$_0$ (any byte-string of 1 to 255 characters), case$_0$ (a previously defined word) will execute; if it equals key$_1$, case$_1$ executes, and so on. The default case will execute on no match; if no default is specified, NOOP is assumed. Cases may be IMMEDIATE words, but no compile-time execution will occur within the case structure. The structure must be terminated by END-CASE. (See END-CASE).

Explanation of Screens by Number

100-102: To enable the loading of a screen, delete the leftmost parenthesis. For all screens above 108, 109 must be loaded. Some screens load others that they require, hence loading all screens will cause some to be loaded twice. If it is not desired to load all the examples, edit DECIMAL ;S on the same line of any screen in which the word (EXAMPLE) appears on a line by itself.

103: END-CASE is an example of a terminator (or a leader, or any structure) that is common to all members of some group (in this instance, casewords). The structure can be identical for all members of the group only because it behaves slightly differently to each of them. END-CASE accomplishes this by following a binary tree. At each node a flag variable is tested and code common to the branch taken is executed. All members of the group (each caseword) must set or reset the flag variables that must be tested to complete the execution of all their compile-time code.

END-CASE must be expanded for each new class of casewords that use it as a common compile-time terminator. This is done by creating a new flag variable that is 1 only for members of the new class. The affected casewords are then amended to set or reset this variable (at compile-time) depending on their membership in the class, and the new variable is tested in END-CASE. So far, I have included only two classes: the unique, indexed CASE, and the keycasewords. Each is further sub-divided into defining word and inline forms. Note that STATE can serve as a flag for this distinction, providing that the case defining word executes outside of a colon definition, and the inline form does not. Instead of using a binary tree (nested IF tests) with a new flag variable required for each branch, consider using a caseword inside END-CASE, itself, to accomplish an n-way branch based on the value of a single variable!

105: CASE is the simplest form of n-way branch. It compiles a string of consecutive codefield addresses (CFA's) exactly like the parameter field of a colon definition. The : on line 3 creates the header and sets compile mode, END-CASE terminates compile mode. Whereas the CFA's in a colon definition execute sequentially, only one CFA will execute each time a CASE is called. It expects an index on the run-time stack; if it is 0 the first CFA executes, if is 1 the second CFA executes, and so on. No protection is made for out-of-range indices. Credit for the basic form of CASE goes to J. B. Weems, also of Hughes Aircraft, Fullerton.

106-107: Each caseword is presented in three forms: a ;CODE defining word, a <BUILDS DOES> defining word, and an inline version. The inline version is perhaps closest to ordinary usage, the <BUILDS DOES> defining casewords are machine independent and easiest to modify, and the ;CODE defining casewords are, in all cases, the fastest. This is because they take advantage of the available system pointer W (which is set by NEXT) in order to index into the parameter field of the case structure; whereas the inline casewords must move IP beyond the case structure after using it to select a case. Note that the inline casewords are not defining words, and so do not require an auxiliary name for the case structure.

The method of putting the CFA to be executed into W and jumping to the last half of NEXT (which fetches the code address and puts it into the program counter), is based on the word EXECUTE as a model. The "NEXT 6 + JMP" used here is source truncation for space purposes. It assumes that no insertions are made in the beginning of NEXT (an insertion in NEXT might be forgivable if short and forbidden to execute at run-time, or if turned on momentarily by an EXEVAR). In such a case, the safe thing (and in any case, the fast thing) to do is to copy the code for the last half of NEXT (however it appears on your machine), rather than jumping to it.

<u>108</u>: A curious hybrid of high-level inside a CODE definition. DO-CASE is really a macro that compiles code similar to that executed by the inline version. Note that if the stack is 0, and DO-CASE executes one of the cases, execution will not return to 3TEST, but to the word calling 3TEST (that is, the HPUSH JMP will not execute). There is no danger of name confusion because the two DO-CASE's are in separate vocabularies.

<u>109-110</u>: A keycase is so called because it requires a key associated with each CFA in the case structure. A key of the same type must be supplied at run-time. If a matches a key in the caseword, the associated CFA will be executed. Unless a match is guaranteed, a default CFA is required which is executed on no match. The default may be a NOOP, a pop of the parameter stack, or even a link to another caseword. The default case is optional, if none is specified, ,CFA compiles a reference to NOOP, automatically.

The structure of a keycaseword is as follows:

$\text{COUNT} \quad \text{KEY}_0 \quad \text{CFA}_0 \quad \text{KEY}_1 \quad \text{CFA}_1$
default

$\ldots \quad \text{KEY}_n \quad \text{CFA}_n \quad \text{CFA}$

Where count is the number of cases (default excluded), and CFA_0 is the CFA that will be executed if the run-time key matches KEY_0. The count is not supplied by the programmer; it is determined automatically by ,KEYCASE by counting the number of cases till END-CASE at compile-time. The HERE 0 C, on line 13 reserves space for the count, and it is filled in by END-CASE. The 1+ after the BEGIN on line 14 is incrementing the case count on the stack. The compiled count will be picked up at run-time to become a DO LOOP index. When the index runs to 0, it indicates that the list of cases is exhausted, and the defualt address is to be followed.

Just under this count, on the stack, is a flag that indicates whether the programmer has not supplied a default case. It starts at 0 on line 13, may be changed to 1 by line 4, and is tested on line 11.

All of the keycasewords, as written, reserve only one byte for the count of the number of cases. Hence, one is limited to 255 cases per case structure (0 is not allowed, either). However, keys need not be consecutive or ordered in any fashion, as are the indices for CASE. Keys may be 1, 2, or n bytes depending on the kind of caseword; CFA's are 2 bytes.

In addition, inline casewords compile a 2-byte address in front of the count. This "exit address" points to the first byte beyond the end of the case structure. This address is put into IP so that execution may resume after a case has executed. Case defining words do not need this because IP already points correctly; W is used to scan the case structure.

The 1 = IF on lines 1 and 9 is testing for NULL (alias X). NULL cannot perform its usual function of resetting IN, incrementng BLK, and terminating the loading of a screen. The reason is that ,CFA uses ' CFA , which is capable of compiling a reference to even an IMMEDIATE word. This has the advantage that an IMMEDIATE word can be called as a case, but no compile-time execution is permitted in the middle of a case structure. Lines 1 and 9 perform part of the definition of NULL if one is detected. Not that the test assumes 8080 byte order; on some machines, the test for NULL would be 0100 = IF. If, on your system, a block equals a screen, all the testing for NULL may be deleted.

,KEYCASE is designed so that one or more keys, CFA's, default, or END-CASE may be on any given line. A key need not even be on the same line as the associated CFA. Do not skip lines in the middle of a case definition. Keys and CFA's must alternate, the exception is the default CFA which has no key.

,XKEY on line 7, is a dummy which is called by ,KEYCASE where it reserves 2 bytes which will be filled in at compile-time when a particular caseword executes (lines 8 and 9 on screen 111, for example). The CONSTANT ,XKEY-ADDR defined on line 6 is set to point to the two bytes reserved in ,KEYCASE so that a reference to a ,KEY appropriate to a given keycaseword may be stored there (by !KEY , for example). A more elegant solution, beyond the scope of this document, would be to make ,XKEY an EXEVAR , a variable whose value is assumed to be a CFA, and which is executed rather than fetched. !1KEY , !2KEY , etc., would then be used to set the EXEVAR to ' ,1KEY CFA or ' ,2KEY CFA.

NOWSAVE and RECOVER are needed because by the time END-CASE is encountered, one has typically already compiled the default case as a key instead of a CFA. This is because it breaks the pattern of KEY CFA , KEY CFA. And in any case, in order to recognize END-CASE, we must advance the input pointer beyond it, and it is convenient to restore it so that END-CASE can execute and perform its compile-time activity. RECOVER is, then, a way to un-compile and un-interpret what has been done.

The endless loop of line 14 is terminated by the R> DROP on line 10 when END-CASE is encountered. This assumes that ,KEYCASE will always call ,CFA directly.

111: Line 10: BYTECASE is a typical 8080 ;CODE defining word. "HEADER !1KEY ,KEYCASE" is the compile-time activity, and the macro RUN-BYTECASE compiles the run-time code. The run-time code must leave W pointing at a case CFA or the default CFA, and then execute that CFA.

Warning: if your ASSEMBLER does not specifically define BEGIN as HERE (non-IMMEDIATE), then you will fall through the ASSEMBLER into FORTH and find : BEGIN HERE ; IMMEDIATE. This version will not work in macros, because you want to compile a reference to BEGIN that will execute when the macro executes.

Note how simply each key is paired with the word to execute upon matching the key (32 TWO, for example). The only punctuation needed is spaces (the number of spaces is not important).

112: It is interesting that 5TEST does not behave exactly like 4TEST. They are designed so that pressing the terminal key "0" selects ;S to be executed. Because <BUILDS DOES> is high-level, it has an extra level on the return stack; hence, the endless loop on line 13 does not exit, but screen 111 returns to the terminal with "OK".

Calling the colon definition "R> R> DROP DROP" from 112 would have the same effect as calling the code ;S from 111.

113: Note that the inline code is 6 instructions longer than the run-time code of the defining word. These instructions pick up the "exit address" which was given space at compile time by the "HERE 0 , "on line 7, and filled in by END-CASE.

114: Same idea as 111, but a two-byte key is expected on the stack. The low byte is in L and high byte is in H. Compare the "," on line 12 with the "C," of 111 line 8.

115-117: Self-explanatory.

118: Stringcase uses variable-length keys (up to 255 bytes). At run-time it expects bytes beginning at HERE 1+ with a count of them at HERE. It will match this string against its keys, executing the associated CFA if a match occurs. There is no restriction that the bytes must be printable ASCII, but you may find it hard to edit anything else into a screen. Source numbers may be used as keys, but they will be treated as character strings; the run-time is also a byte string, it is not normally placed on the stack even if it is a number.

The run-time code has two loops. The outer one is counting down the number of cases; the inner one has an index equal to the byte-count of a key plus one (the count, itself, is compared). Saved on the stack is IP and the address of the next key in the case structure (computed from W plus the byte-count plus 3)

119: One application of STRINGCASE is as a compact language translator. The string key is the input word, and the word executed by the associated CFA is the translation. Such an association is faster than a colon definition equating the two, because IP is not saved on the return stack, or restored.

The cases of a stringcase constitute a sort of vocabulary, but the structure is more compact than an ordinary dictionary because it lacks link fields, code fields, and terminators. The arithmetic that advances W from one case to the next is almost as fast as following a dictionary link, and the code for RUN-STRINGCASE compares favorably with (FIND). It is hard to imagine a FORTH - like language translator that would be faster or more compact.

120: High-level version of 118. The two nested loops are still there as DO LOOPS, the address of the next case is saved on the return stack between the loops, and the two pointers to the two byte-strings are on the parameter stack.

121-122: Self-explanatory.

123: The word called by the default case is exactly like INTERPRET except that is does not need to do a BL WORD because the string is already at HERE, and it is not an endless loop (so that it INTERPRET's only one word).

GERMAN is, then, a STRINGCASE that will, first, attempt to translate a word, but if it is not in its vocabulary, it will INTERPRET it normally. The endless loop taken away from INTERPRET is given to TRANSLATE which is then substituted for INTERPRET in the definition of LOAD (it could also be substituted in the definition of QUIT).

If one now loads a screen with TLOAD, it should compile normally, with the addition that EIN, ZWEI, and DREI will be understood as re-names, and executed immediately. In order to be a true translating interpreter, the DOES> part of STRINGCASE must be extended to respect compile mode by testing STATE, and either compile the CFA or execute the CFA, depending.

Note that ;S calls itself as a case. This is not only a way to find ;S, the interpreter would not stop at either ;S or NULL (it would, of course, stop at an undefined). The reason is that there is not an extra level on the return stack (namely, TRANSLATE) between the equivalent of LOAD (TLOAD), and the equivalent of INTERPRET (DEFAULT). Hence, executing ;S from DEFAULT is sufficient to end the execution of GERMAN, but not of TRANSLATE (which will inevitably call GERMAN again). However, calling ;S as a case, since it is a CODE definition, will end the execution of TRANSLATE, and return eventually to the terminal with "OK". But if one had used the <BUILDS DOES> version of STRINGCASE, there would, again, be an extra level on the return stack, and ;S would again fail. In this case, ;S would have to call a word whose definition is R> R> DROP DROP (see explanation of screen 112).

Another possible kind of keycase might be called BITCASE, where the key is a mask, and the associated CFA executes if the mask AND'ed with the value on the stack ≠ 0. The flag variables and compile-time code would be identical with BYTECASE; the run-time code would simply do an AND instead of an XOR, and a 0 = NOT instead of 0 = . The casewords presented here by no means exhaust the possibilities. The structure is deliberately left open-ended to encourage user creativity.

Note that BITCASE, BYTECASE, 2BYTECASE, and STRINGCASE all differ in name-length to avoid confusion on WIDTH-3 systems even when prefixed by DO- or RUN- .

Keycases have the property that they can be chained together through their default addresses (the key can be changed at this point, as well). This makes possible complex, high-level sturctures in which casewords feed other kinds of casewords. This is a tree with n brances at each node (a pattern similar to human brain cells).

With two default CFA's one could put keycasewords into 2-link structures such as binary trees. Furthermore, any CFA, including the default case, can be an EXEVAR (see explanation of screen 109), allowing the structure of the tree to change dynamically at run-time.

;S Steve Munson
Hughes Aircraft Company
Fullerton, CA 92634

Judges' Comments - An interesting aproach to error control, by making : IMMEDIATE a part of error control. If a preceding ; is missing, due to misediting, : will be encountered in compile mode. It executes but contains ?EXEC, which produces an error message if compiling. A little confusing but it works.

100

```
 0 ( LOAD SCREEN LOADER FOR CASEWORD SELECTION )
 1 FORGET TASK    : TASK ;
 2
 3   103 LOAD    ( END-CASE)
 4   109 LOAD    ( WORDS FOR KEYCASEWORD COMPILING)    ( LOADS 110)
 5
 6 ( 101 LOAD    ( LOAD SCREEN #1)
 7   102 LOAD    ( LOAD SCREEN #2)
 8 ;S
 9
10
11
12
13
14
15
```

101

```
 0     ( LOAD SCREEN #1 FOR CASEWORD SELECTION )
 1
 2 ( 105 LOAD  ( CASE DEFINING WORD)
 3   106 LOAD  ( <BUILDS DOES> CASE DEFINING WORD)
 4 ( 107 LOAD  ( IN-LINE CASE WORD)
 5   108 LOAD  ( IN-LINE CASE FOR CODE DEFINITIONS)
 6
 7 ( 111 LOAD  ( BYTECASE DEFINING WORD)
 8 ( 112 LOAD  ( <BUILDS DOES> BYTECASE DEFINING WORD)
 9 ( 113 LOAD  ( IN-LINE BYTECASE WORD)                ( LOADS 111)
10
11   114 LOAD  ( 2BYTECASE DEFINING WORD)              ( LOADS 115)
12 ( 116 LOAD  ( <BUILDS DOES> 2BYTECASE DEFINING WORD)
13 ( 117 LOAD  ( IN-LINE 2BYTECASE WORD)               ( LOADS 114)
14 ;S
15
```

102

```
 0     ( LOAD SCREEN #2 FOR CASEWORD SELECTION )
 1
 2   118 LOAD  ( STRINGCASE DEFINING WORD)             ( LOADS 119)
 3   120 LOAD  ( <BUILDS DOES> STRINGCASE DEFINING WORD  ( "   121)
 4 ( 122 LOAD  ( IN-LINE STRINGCASE WORD)              ( LOADS 123)
 5 ;S
 6
 7
 8
 9
10
11
12
13
14
15
```

103

```
 0     ( END-CASE)  HEX
 1 0 VARIABLE ?INLINE    0 VARIABLE ?KEYCASE
 2
 3 : +INLINE   1 ?INLINE  ! ;      : -INLINE   0 ?INLINE  ! ;
 4 : +KEYCASE  1 ?KEYCASE ! ;      : -KEYCASE  0 ?KEYCASE ! ;
 5
 6 : END-CASE   ?KEYCASE @ IF  SWAP C!                 ( UNIVERSAL)
 7     STATE @ IF  HERE SWAP !  ENDIF  ELSE            ( CASEWORD )
 8   ?INLINE @ IF  HERE SWAP !  ELSE SMUDGE            ( DELIMITER)
 9        [COMPILE] [  ENDIF  ENDIF ;   IMMEDIATE
10 DECIMAL ;S
11
12
13
14
15
```

105

```
 0     ( CASE DEFINING WORD )    HEX
 1
 2 : CASE  -KEYCASE -INLINE             ( SET FLAG VARIABLES FOR CASE )
 3     [COMPILE]    ;CODE               ( DEFINE CASE WORD   COMP CFAs)
 4   H POP,  H DAD,  W INX,  W DAD,  ( CODE EXECUTES WORD VIA INDEX)
 5   M W 1+ MOV,  H INX,  M W MOV,   ( ON STACK.    PUT CFA INTO W . )
 6   XCHG,  NEXT 6 + JMP,             ( JUMP TO LAST HALF OF NEXT)
 7
 8                        ( EXAMPLE )
 9 : ONE  1  . ;      : TWO  2  . ;      : THREE  3  . ;
10
11 CASE PICK  ONE TWO THREE  END-CASE
12
13 : 0TEST   0 .    PICK  4 . ;             ( 0 0TEST PRINTS 0 1 4 )
14 DECIMAL ;S                              ( 1 0TEST PRINTS 0 2 4 )
15                                         ( 2 0TEST PRINTS 0 3 4
```

106

```
 0     ( <BUILDS DOES> CASE DEFINING WORD )    HEX
 1
 2 : CASE   <BUILDS  -KEYCASE -INLINE  ]        ( ADD INDEX TO PFA)
 3             DOES>  OVER + +  @ EXECUTE ;     ( GET CFA, EXECUTE)
 4
 5                        ( EXAMPLE )
 6 : ONE  1  . ;     : TWO  2 . ;     : THREE  3 . ;
 7
 8 CASE PICK  ONE TWO THREE  END-CASE
 9
10 : 1TEST   0 .    PICK  4 . ;
11 DECIMAL ;S
12
13
14
15
```

107

```
 0     ( IN-LINE CASE WORD )    HEX
 1
 2 CODE DO-CASE   IP H MOV,   IP 1+ L MOV,      ( PICK UP EXIT ADDR )
 3    M IP 1+ MOV,  H INX,  M IP MOV,    ( MOVE IP BEYOND END-CASE)
 4    XCHG,  H POP,  H DAD,  W INX,  W DAD,    ( H = W + 2* OFFSET)
 5    M W 1+ MOV,  H INX,  M W MOV,    ( CODE FIELD ADDR  INTO W  )
 6    XCHG,  NEXT 6 + JMP,              ( JUMP TO LAST HALF OF NEXT)
 7
 8 : DO-CASE  -KEYCASE +INLINE                     ( HERE 0   LEAVES)
 9    COMPILE DO-CASE  HERE 0 , ;   IMMEDIATE      ( GAP    END-CASE)
10                                                 ( FILLS IT IN   )
11                        ( EXAMPLE )
12 : ONE  1  . ;      : TWO  2  . ;      : THREE  3  . ;
13
14 : 2TEST   0 .   DO-CASE  ONE TWO THREE  END-CASE  4 . ;
15 DECIMAL ;S
```

108

```
 0     ( IN-LINE CASE FOR CODE DEFINITONS )    HEX
 1 ASSEMBLER DEFINITIONS
 2 : DO-CASE   HERE D - W LXI,          ( LOAD W WITH ADDR BEYOND LAST
 3    H POP,  H DAD,  W DAD,            ( BYTE OF THIS MACRO , ADD 2* )
 4    M W 1+ MOV,  H INX,  M W MOV,  ( INDEX ON STACK , FETCH CFA   )
 5    XCHG,  NEXT 6 + JMP,              ( JUMP TO LAST HALF OF NEXT   )
 6    -KEYCASE -INLINE  SMUDGE  ] ;  ( SET FLAGS AND COMPILE MODE   )
 7 FORTH DEFINITIONS
 8
 9                        ( EXAMPLE )
10 : ONE  1 . ;      : TWO  2  . ;      : THREE  3  . ;
11                                               ( 1 3TEST   PRINTS 1)
12 CODE 3TEST   H POP,  L A MOV,  H ORA,        ( 0 0 3TEST PRINTS 1)
13    0= IF,  DO-CASE  ONE TWO THREE  END-CASE  THEN,  HPUSH JMP,
14 DECIMAL ;S
15
```

109

```
 0     ( WORDS FOR KEYCASEWORD COMPILING )     HEX
 1 0 VARIABLE OLDHERE    0 VARIABLE OLDBLK    0 VARIABLE OLDIN
 2
 3 : NOWSAVE   HERE OLDHERE !  BLK @ OLDBLK !   IN @ OLDIN !
 4   RECOVER   OLDHERE @ DP !  OLDIN @ IN !  OLDBLK @ BLK ! ;
 5
 6 : *NUMBER   -1 DPL !   0 0 HERE  DUP 1+ C@ 2D =
 7    DUP >R +  (NUMBER) DROP DROP  R> IF  MINUS  ENDIF ;
 8
 9 : HEADER   CREATE SMUDGE ;
10
11 : *'   -FIND  0= 0 ?ERROR  DROP ;           (   WITHOUT LITERAL)
12 DECIMAL -->
13
14
15
```

110

```
 0     ( WORDS FOR KEYCASEWORD COMPILING, CONT. )  HEX
 1 : ?KEY   -FIND  HERE @ 1 = IF  1 BLK +!  0 IN !      ( IF NULL  )
 2    NOWSAVE  DROP DROP DROP -FIND ENDIF  ( GET NEXT BLOCK, SAVE)
 3       IF  DROP  ' END-CASE = IF              ( NEW ADDR. END-CASE?)
 4    SWAP 1+ SWAP  RECOVER ENDIF ENDIF ;      ( SET NO-DEFAULT FLAG )
 5
 6 0 CONSTANT ,XKEY-ADDR                     ( ADDR OF ,XKEY IN KEYCASE)
 7 : ,XKEY  HERE 0 ,     ' ,XKEY-ADDR ! ;  IMMEDIATE ( SET ,XKEY-ADDR)
 8
 9 : ,CFA   *'  HERE @ 1 = IF  1 BLK +!  0 IN !  DROP *'   ( PUT A)
10   ENDIF  DUP ' END-CASE = IF  RECOVER  R> DROP  DROP   ( CFA IN)
11      SWAP IF  ' NOOP ELSE  *'  ENDIF ENDIF  CFA , ;    ( A CASE)
12
13 : ,KEYCASE   +KEYCASE  HERE 0 C,  0
14   -1 BEGIN  1+  NOWSAVE  ?KEY  ,XKEY  ,CFA  AGAIN ;
15 DECIMAL ;S
```

111

```
 0     ( BYTECASE DEFINING WORD, ( ONE-BYTE KEYS )     HEX
 1 ASSEMBLER DEFINITIONS
 2 : RUN-BYTECASE   H POP,  W INX,  W LDAX,  A H MOV,  W INX,
 3    BEGIN, W LDAX,  W INX,  L XRA,  0= NOT IF,       ( MACRO USED)
 4            W INX,  W INX,  H DCR,  THEN,  0= END,    ( ON LINE 10)
 5    XCHG,  M W 1+ MOV,  H INX,  M W MOV,              ( & SCR 113, )
 6    XCHG,  NEXT 6 + JMP, ;    FORTH DEFINITIONS       ( LINE 4    )
 7
 8 : ,1KEY   *NUMBER C, ;                    ( COMPILE A 1-BYTE KEY )
 9 : !1KEY   ' ,1KEY CFA  ,XKEY-ADDR ! ;    ( PUT ,1KEY IN KEYCASE)
10 : BYTECASE   HEADER  !1KEY ,KEYCASE  ;CODE RUN-BYTECASE
11                        ( EXAMPLE )
12 : ONE  1  . ;     : TWO  2 . ;     : THREE  3 . ;     : DEFAULT 4 . ;
13 BYTECASE SHOW
14    31 ONE  30 ;S  32 TWO  33 THREE  DEFAULT  END-CASE
15 : 4TEST   BEGIN  KEY SHOW  AGAIN ;    DECIMAL
```

112

```
 0     ( <BUILDS DOES> BYTECASE DEFINING WORD )     HEX
 1 : ,1KEY   *NUMBER C, ;                    ( COMPILE A 1-BYTE KEY )
 2 : !1KEY   ' ,1KEY CFA  ,XKEY-ADDR ! ;    ( PUT ,1KEY IN ,KEYCASE)
 3
 4 : BYTECASE   <BUILDS  !1KEY ,KEYCASE  DOES>
 5    DUP C@  0 DO  1+ OVER OVER  C@ = IF            ( HIGH LEVEL DOES)
 6       LEAVE ELSE  2+ ENDIF  LOOP                  ( SAME  THING  AS)
 7    SWAP DROP  1+ @ EXECUTE ;                      ( CODE ON SCR 111
 8
 9                        ( EXAMPLE )
10 : ONE  1  . ;     : TWO  2 . ;     : THREE  3 . ;     : DEFAULT 4 . ;
11 BYTECASE *SHOW
12    31 ONE  30 ;S  32 TWO  33 THREE  DEFAULT  END-CASE
13 : 5TEST   BEGIN  KEY *SHOW  AGAIN ;
14 DECIMAL ;S
15
```

113

```
    ( IN-LINE BYTECASE WORD )
111 LOAD   HEX

CODE DO-BYTECASE   IP H MOV.  IP 1+ L MOV.  M IP 1+ MOV.
   H INX.  M IP MOV.  XCHG.  RUN-BYTECASE  ( SEE MACRO SCR 111)

: DO-BYTECASE
   COMPILE DO-BYTECASE  HERE 0 ,   '1KEY ,KEYCASE ;   IMMEDIATE

                         ( EXAMPLE )
  : ONE  1 . ;       : TWO  2 . ;      : THREE  3 . ;      : DEFAULT 4 . ;

: 8TEST   BEGIN  KEY  DO-BYTECASE  31 ONE  32 TWO
   33 THREE  30 ;S  DEFAULT  END-CASE  AGAIN ;
DECIMAL ;S

```

114

```
    ( 2BYTECASE DEFINING WORD, ( TWO-BYTE KEYS )     HEX
ASSEMBLER DEFINITIONS                                ( H = KEY ,)
  RUN-2BYTECASE   H POP.   IP PUSH.                  ( SAVE IP ,)
   W INX.  W LDAX.  A IP MOV.  W INX.             ( IP = # OF CASES)
   BEGIN.  W LDAX.  W INX.  L XRA.  0= IF.           ( 1ST BYTE = ?)
           W LDAX.  W INX.  H XRA.  0= NOT IF.       ( 2ND BYTE = ?)
           W INX.  W INX.  IP DCR.  THEN.  ELSE.  ( IF NO MATCH,)
           W INX.  W INX.  W INX.  IP DCR.  THEN.  ( GO NEXT CASE)
   0= END.  IP POP.                                  ( ELSE GET IP.)
   XCHG.  M W 1+ MOV.  H INX.  M W MOV.              ( PICK UP CFA.)
   XCHG.  NEXT 6 + JMP. ;    FORTH DEFINITIONS       ( EXECUTE CFA )

: ,2KEY   *NUMBER , ;                       ( COMPILE A 2-BYTE KEY )
: '2KEY      ,2KEY CFA  ,XKEY-ADDR ' , ;    ( PUT ,2KEY IN KEYCASE)
: 2BYTECASE   HEADER  '2KEY ,KEYCASE  ;CODE RUN-2BYTECASE
DECIMAL -->
```

115

```
    ( 2BYTECASE DEFINING WORD, CONT )    HEX
( TEST FOR SCREEN 114)
                         ( EXAMPLE )
  : ONE  1 . ;       : TWO  2 . ;      : THREE  3 . ;      : DEFAULT 4 . ;

2BYTECASE 7PICK
   0111 ONE  0222 TWO  0333 THREE  DEFAULT  END-CASE
: 7TEST   0     7PICK  4  ;
DECIMAL ;S







```

116

```
    ( <BUILDS DOES> 2BYTECASE DEFINING WORD )    HEX
: ,2KEY   *NUMBER , ;                       ( COMPILE A 2-BYTE KEY )
: '2KEY      ,2KEY CFA  ,XKEY-ADDR ' , ;    ( PUT ,2KEY IN ,KEYCASE)

: 2BYTECASE   <BUILDS  '2KEY ,KEYCASE  DOES> ( DOES SAME THING )
   DUP 1+ SWAP C@  0 DO  OVER OVER  @ = IF    ( AS CODE ON 114, )
      2+ LEAVE ELSE  4 + ENDIF  LOOP          ( COMPARES BOTH )
   SWAP DROP  @ EXECUTE ;                     (  BYTES AT ONCE )

                         ( EXAMPLE )
  : ONE  1 . ;       : TWO  2 . ;      : THREE  3 . ;      : DEFAULT 4 . ;

2BYTECASE 8PICK
   0111 ONE  0222 TWO  0333 THREE  DEFAULT  END-CASE
: 8TEST   0     8PICK  4  ;
DECIMAL ;S
```

117

```
    ( IN-LINE 2BYTECASE WORD )
114 LOAD   HEX

CODE DO-2BYTECASE   IP H MOV.  IP 1+ L MOV.  M IP 1+ MOV.
   H INX.  M IP MOV.  XCHG.  RUN-2BYTECASE ( SEE MACRO SCR 114)

: DO-2BYTECASE
   COMPILE DO-2BYTECASE  HERE 0 ,   '2KEY ,KEYCASE ;   IMMEDIATE

                         ( EXAMPLE )
  : ONE  1 . ;     : TWO  2 . ;     : THREE  3 . ;     : DEFAULT 4 . ;

: 9TEST   0    DO-2BYTECASE  0111 ONE  0222 TWO
   0333 THREE  0 ;S  DEFAULT  END-CASE  4  ;
DECIMAL ;S

```

118

```
    ( STRINGCASE DEFINING WORD, ( STRING KEYS )     HEX
ASSEMBLER DEFINITIONS
  RUN-STRINGCASE   IP PUSH.  W INX.               ( SAVE IP ON STACK)
           W LDAX.  A IP MOV.  W INX.              ( IP = # OF CASES
   BEGIN.  W LDAX.  A INR.  A IP 1+ MOV.  W 1+ ADD.    ( IP 1+ =)
           A L MOV.  W A MOV.  0 # ADC.  A H MOV.  ( BYTE COUNT)
           H INX.  H INX.  H PUSH.            ( STACK = NEXT CASE)
           DP ( RUNTIME HERE) LHLD.            ( H = RUN-TIME HERE)
   BEGIN.  W LDAX.  W INX.  M XRA.  H INX.  0= IF.  ( BYTES = ?)
           IP 1+ DCR.  SWAP ( AT COMPILE TIME) 0= END.  ( AGAIN)
           H POP.  HERE 8 + JMP.  THEN.      ( THROW AWAY ADDR )
           W POP.  IP DCR. 0= END.  IP POP. ( ELSE GO NEXT CASE)
   XCHG.  M W 1+ MOV.  H INX.  M W MOV.            ( PICK UP CFA )
   XCHG.  NEXT 6 + JMP ;   FORTH DEFINITIONS       ( JUMP TO NEXT)
DECIMAL -->

```

119

```
    ( STRINGCASE DEFINING WORD, CONT )    HEX

: ,SKEY   HERE C@ 1+  ALLOT ;             ( COMPILE A STRING KEY )
: 'SKEY      ,SKEY CFA  ,XKEY-ADDR ' ;     ( PUT ,SKEY IN ,KEYCASE)

: STRINGCASE   HEADER  'SKEY ,KEYCASE  ;CODE RUN-STRINGCASE

                         ( EXAMPLE )
  : ONE  1 ;         : TWO  2 . ;      : THREE  3 ;        : DEFAULT 4
                    ( DECIMAL 122 LOAD ;S
STRINGCASE GERMAN
   EIN ONE  ZWEI TWO  DREI THREE  DEFAULT  END-CASE

: TRANSLATE   BL WORD  GERMAN      ( "TRANSLATE EIN" PRINTS 1)
DECIMAL ;S

```

120

```
    ( <BUILDS DOES> STRINGCASE DEFINING WORD )    HEX
: ,SKEY   HERE C@ 1+  ALLOT ;             ( COMPILE A STRING KEY )
: 'SKEY   ' ,SKEY CFA  ,XKEY-ADDR ' , ;   ( PUT ,SKEY IN ,KEYCASE)

: STRINGCASE   <BUILDS  'SKEY ,KEYCASE  DOES>
  HERE OVER 1+  ROT C@ 0 DO                     ( DO # OF CASES )
     DUP DUP C@ + 3 + >R  DUP C@ 1+ 0 DO       ( SAVE NEXT CASE)
     OVER C@  OVER C@ = IF  1+ SWAP  1+ SWAP  (  COMPARE BYTE )
     ELSE  DROP DROP HERE 0  LEAVE ENDIF LOOP   ( IF MATCH  DUMP )
   DUP IF  R> DROP  LEAVE ELSE  DROP R>  ENDIF    ADDR , ELSE GO
   LOOP  SWAP DROP  @ EXECUTE ;                 ( TO NEXT CASE  )
DECIMAL -->




```

121

```
    ( <BUILDS DOES> STRINGCASE DEFINING WORD, CONT )    HEX
( TEST FOR SCREEN 120)
                         ( EXAMPLE )
  : ONE  1 . ;       : TWO  2 . ;      : THREE  3 . ;      : DEFAULT 4 . ;

STRINGCASE GERMAN
   EIN ONE  ZWEI TWO  DREI THREE  DEFAULT  END-CASE

: TRANSLATE   BL WORD  GERMAN ;
DECIMAL ;S






```

122

```
    ( IN-LINE STRINGCASE WORD )
118 LOAD   HEX

CODE DO-STRINGCASE   IP H MOV   IP 1+ L MOV.  M IP 1+ MOV.
  H INX.  M IP MOV.  XCHG.  RUN-STRINGCASE ( SEE MACRO SCR 118)

: DO-STRINGCASE
   COMPILE DO-STRINGCASE  HERE 0 ,   'SKEY ,KEYCASE ;   IMMEDIATE

                         ( EXAMPLE )
  : ONE  1 . ;       : TWO  2 . ;      : THREE  3 . ;      : DEFAULT 4 . ;

: TRANSLATE   BL WORD  DO-STRINGCASE
   EIN ONE  ZWEI TWO  DREI THREE  DEFAULT  END-CASE ;
DECIMAL ;S

```

123

```
    ( EXAMPLE OF A TRANSLATING INTERPRETER)    HEX
: DEFAULT   HERE CONTEXT @ @ (FIND)  DUP 0= IF      ( = INTERPRET)
      DROP  HERE LATEST (FIND)  ENDIF       ( WITHOUT BEGIN-AGAIN)
  IF  STATE @ < IF  CFA ,  ELSE  CFA EXECUTE  ENDIF ?STACK
  ELSE  HERE NUMBER  DPL @ 1+ IF  [COMPILE] DLITERAL      ALSO )
        ELSE  DROP  [COMPILE] LITERAL  ENDIF ?STACK       ( LACKS)
  ENDIF ;                                                 ( BL WORD)

STRINGCASE GERMAN
   EIN ONE  ZWEI TWO  DREI THREE  .S ;S  DEFAULT  END-CASE

: TRANSLATE   BEGIN  BL WORD  GERMAN  AGAIN ;

: TLOAD   BLK @ >R  IN @ >R                      ( SAME AS LOAD, )
   0 IN !  B/SCR *  BLK !                        ( BUT TRANSLATE)
      TRANSLATE  R> IN !  R> BLK !    DECIMAL   FOR INTERPRET)
```

A PROPOSED CASE STATEMENT FOR FORTH

Karl Bochert/Dave Lion

General Description

The CASE statement suggested here is done in high level code for the 6800 version of fig-FORTH. It may have to have some minor changes in order to conform to the FORTH-79 standard. The names of the words were chosen for descriptive value.

The word that initiates the set of cases is:

```
CASE
```

Following that are as many sets of:

```
<forth code>  ENDCASE
```

as needed to represent all the desired cases which are to be executed. The first set is for case 0, and each successive set is for the next higher case number. After the last set comes the terminating set:

```
<forth code>  ENDCASES
```

which indicates the default code to be executed if the case number is outside the legal limits. It also marks the end of all of the cases, and causes the look-up table to be compiled. Word (CASE) , which is the run-time word, is surrounded by parentheses according to fig-FORTH convention, indicating that it is normally never typed in by the user.

At run-time word (CASE) uses one integer parameter from the data stack and leaves none. The given parameter specifies which one of many cases will be executed. A single case is defined as a set of FORTH words which is preceded by the word CASE or ENDCASE, and followed by the word ENDCASE or ENDCASES. Within a single case, the usual rules of pairing still apply to the words: DO, LOOP, IF, ELSE, THEN, BEGIN, AGAIN, WHILE, REPEAT. That is, they must be properly matched with each other.

Case 0 will be executed if the parameter is 0, case 1 if it is 1, etc. The parameter will normally be in the range: 0 thru (# of cases)-1. Thus, the case function works like the computed GOTO found in some versions of BASIC, with the exception that this code is in-line.

Advantages

CASE is very compactly compiled, so the number of 16-bit words of overhead is 2 * (# of cases +1) + 3. This excludes the code within each of the cases, but includes the ;S which follows each case. The following use of the CASE function, having 3 empty cases and an empty default case will compile as 22 bytes of code:

```
CASE
   ENDCASE
   ENDCASE
   ENDCASE
ENDCASES
```

Here, it should be pointed out that the CASE function is only used within a definition, and the above sample is part of a definition.

More Advantages

CASE statements have little overhead run-time code. In the FIG model this version of (CASE) executes 41 FORTH words, 37 of which are code words. This may be shortened by leaving out the two protective features, thus executing 25 words, 22 of which are code words. The fastest method takes about 0.002 seconds to execute.

There is practically no limit upon the number of cases that may be compiled. The table of pointers will contain an address for each case plus an address for the default case.

Two protective functions in word (CASE) will handle negative numbers and numbers that are too high. For negative numbers, the equivalent positive case is executed. For numbers too high, a default case is executed. It should also be noted that any intermediate case that will never be executed still needs an ENDCASE, but the compiled code will contain only a ;S . The default case may be left out, and will then compile like an empty case.

One additional feature to point out is that CASE statements may be nested much the same way as 'DO' loops can.

Disadvantages

There is one machine dependent factor that must be considered before installing these words. Since we fool around with return addresses in the return stack, we must know whether the return stack of the machine stacks 'return to' addresses or 'came from' addresses. The former is the situation where the address is not incremented before doing the first fetch after a ;S . The latter type of machine (my 6800 version) does do a pre-increment after a ;S . Appropriate comments for patching are included in the definition of (CASE) .

The way to find out which type of FORTH machine you are using is:

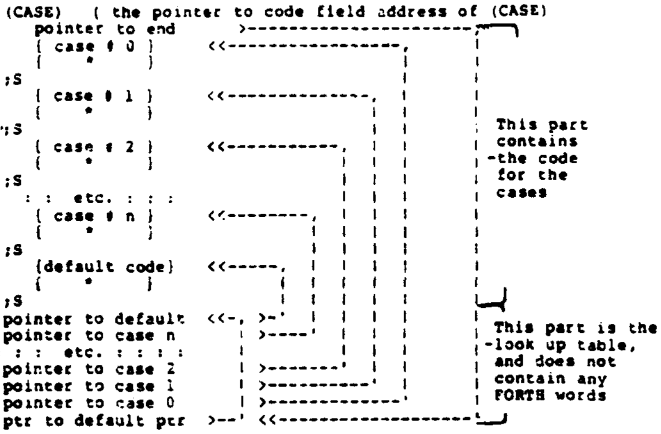
```
: P1  R   ;
: P2  P1  ;
  P2  P2  -  .
  FORGET P1
```

The printout will be 0 for the 'came from' type of FORTH machine, and 2 for the 'return to' type.

Another thing to watch out for is that while inside a CASE statement you no longer have access to any loop counter (I) which was created outside the CASE statement. During execution of the chosen case there is one extra address on the return stack, covering up what was there.

Compiled Structure

Note: Each line shows the contents of one 16-bit word of memory except for the lines within braces: , which signify any amount of memory, including none, which may contain FORTH instructions.



* Any case code may be left out. The resultant case segment will have only a ;S in it.

Definitions for 6800 Fig-FORTH

```
(CASE)                    (    the run-time function                       )
   ABS                    ( #  make sure parameter is +                    )
   R>                     (    get address of pointer to table             )
   2+                ( delete this line for 'return to' machines <======)
   @ DUP                  (    get pointer to table                        )
   2+                  add this line for 'return to' machines <=========)
   >R                     (    save final return address                   )
   SWAP 1+ DUP +          (    find addresses into table of the ....       )
   OVER  SWAP             ( #    highest legal case, ....                  )
   -                      (      and the desired case ....                 )
   SWAP @                 ( #    then choose the ....                      )
   MAX                    ( #    best one                                  )
   @                      (    read table entry for chosen case            )
   2  -              ( delete this line for 'return to' machines <======)
   >R  ;                  (    stack it & 'return' to it                   )

NOTE: the lines marked '#' may be deleted to speed up execution     )
      while sacrificing protection.                                  )


CASE
   COMPILE  (CASE)     (    compile the run-time executor            )
   HERE  0  ,          (    init table pointer & get its addr        )
   0                   (    stack a marker on data stack             )
   ;   IMMEDIATE


ENDCASE
   COMPILE  ;S         ( end of a case                               )
   HERE                ( stack a ptr. to next case                   )
   ;   IMMEDIATE


ENDCASES               ( this word writes the look-up table          )
   COMPILE  ;S         ( end of default case                         )
   ,                   ( put pointer for default case into table     )
   HERE  >R            ( temporarily save addr of pntr to case[n]    )
   DUP                 ( look for case[0]                            )
   IF                  ( didn't find marker, so:                     )
      BEGIN
      ,  DUP  0=       ( store pntrs to case[n] thru case[1]         )
      END              ( until reaching the marker                   )
   THEN
   DROP                ( drop the marker                             )
   DUP                 ( dup the pfa                                 )
   2+  ,               ( store pntr to case[0]                       )
                       ( data stack is down to 1 item: the pfa       )
   HERE SWAP  !        ( store this addr into pfa                    )
   R>                  ( fetch addr of ptr to highest normal case    )
   2  -  ,
   ;   IMMEDIATE
```

A Test of the 'CASE' Function:

```
TEST-WORD
 CR
 5  -4    ( try a range of parameters, some of which are illegal )
 DO
   I  DUP .    ( preceed each line with the case #  being tried )
   CASE
   ." This is the case # 0 code"     ENDCASE
   ( ---case #1 does nothing--- )    ENDCASE
   ." This is the case # 2 code"     ENDCASE
   ." default case"
   ENDCASES
   CR           ( do the next case on a new line )
 LOOP ;
```

The Result is:

```
TEST-WORD   ( typed by human)
-4 default case
-3 default case
-2 This is the case # 2 code
-1
0 This is the case # 0 code
1
2 This is the case # 2 code
3 default case
4 default case
 OK
```

Time Trials:

Here we find out how long it takes to get to the proper case. The CPU clock is set at 1.000 MHz. The word (CASE) was defined leaving out the protection features. Then the following definitions for timing loops are tried, executing null cases which do nothing. 100,000 loops are timed:

```
DECIMAL
: INNER 1000 0 DO 1 CASE ENDCASE ENDCASE ENDCASE ENDCASES LOOP ;
: SPEED ." X" 100 0 DO INNER LOOP ." X" ;
SPEED XX OK     ( this was 210 seconds on the 6800 FORTH )
```

1000,000 loops are timed, leaving out the CASE portion:

```
: INNER2 1000 0 DO        LOOP ;
: SPEED 2 ." X" 100 0 DO  INNER2  LOOP ." X" ;
SPEED2 XX OK     ( this was 13 seconds on the 6800 FORTH )
```

Thus, it can be seen that it takes about 2 milliseconds to vector to the desired case if the two protection features are left out. Putting in the protection would increase the time to about 3.5 mSec.

Karl Bochert
Dave Lion
Los Altos, CA 94022

Judges' Comments -

Karl and Dave were the only entry to make provision for pre-incrementing and post-incrementing versions of NEXT. This refers to when the interpretive pointer IP is advanced within NEXT. They give a test to check your system. This version uses a compiled table of indexes to give minimum execution time. The style and documentation is to be complimented.

CASE AND PROD CONSTRUCTS

Steve Brecher

```
( syntax:   www CASEOF
              www CASE www ESAC
                     .                  ( 0 or more CASE/ESAC pairs
                     .                    allowed, at least 2 pairs
                     .                    for semantic sense.)
              www CASE www ESAC
              OTHERWISE www              ( OTHERWISE optional)
              ENDCASEOF
```

www stands for 0+ Forth words, possibly including complete case expression[s], these possibly still further nested. But code represented by www can make no net change to the return stack, as the case selector value is stored there. Runtime: CASEOF pops, saves top of compute stack as selector. CASE pops, tests top of stack vs. selector; if =, executes words up to next ESAC followed by words after ENDCASEOF. If <>, executes words after next ESAC. OTHERWISE is optional for readability. SELECTOR used anywhere between CASEOF and ENDCASEOF leaves the selector value, provided no net change has been made to the return stack since CASEOF; SELECTOR is an alias for 'R'.)

```
31 CONSTANT      CASSYNTX (Error
                 number, case construct
                 syntax)

: CASEOF

( -> 0 4  .  Pronounced "case of", after Pascal.)
COMPILE >R        ( to save selector for testing by CASEs)
0                 ( end-of-data signal to ENDCASEOF)
4                 ( For CASE syntax check) ;

IMMEDIATE

CODE CASEBRANCH  ( n -> .  Forth branch
                 to the offset
                 following inline if
                 n <> @RP, else bump
                 IP over offset.
                 Compiled by CASE.)
                 S )+   RP () CMP,

NE IF,                        ( If n <> @RP, )
  IP ()   IP ADD,             ( add inline offset to IP)
  NEXT, ENDIF,                ( and "branch")
IP )+ TST,                    ( else bump IP over inline offset)
NEXT, C;                      ( and continue there.)
```

```
: ?CASE          ( n1 n2 -> . Compile-
                   time check for
                   n1=n2. If fail
                   issue syntax error)
                 <> IF CASSYNTX ERROR
                 ENDIF ;

: CASE             ( 4 -> addr 5 .
                   Executes ?CASE
                   syntax check;
                   compiles CASEBRANCH
                   with a zero offset;
                   pushes address of
                   offset so ESAC can
                   fix it later; pushes
                   5 syntax check
                   signal.)
                     4 ?CASE ( Syntax check)
                 COMPILE CASEBRANCH

HERE     ( Push address of offset so ESAC can patch it)
0 ,      ( ESAC will change the 0 to +offset for CASEBRANCH)
5        ( For ESAC syntax check) ;

IMMEDIATE

: ESAC             ( addr1 5 -> addr2
                   4 . Pronounced
                   "eesack"; "case"
                   spelled backward.
                   Executes ?CASE
                   syntax check; fixes
                   the offset at addr1
                   so the CASEBRANCH
                   there will branch to
                   the code after ESAC;
                   compiles BRANCH with
                   a 0 offset, pushes
                   the address of the 0
                   offset so ENDCASEOF
                   can fix it later;
                   leaves 4 for syntax
                   check by later
                   word.)

5 ?CASE                 ( Syntax check)
2                       ( ELSE will be checking for this)
[COMPILE] ELSE          ( ELSE fixes CASE offset, pushes addr
                          of 0 offset it compiles with BRANCH)
2+ ;    ( ELSE leaves 2, CASE/OTHERWISE/ENDCASEOF want 4)

IMMEDIATE
```

```
: OTHERWISE      ( 4 -> 4 .  For
                   readability,
                   optionally written
                   after last ESAC to
                   identify code which
                   is executed if no
                   cases match.
                   Performs compile-
                   time checks.)
                 ?COMP
                 4 ?CASE
                 4 ;
IMMEDIATE

: ENDCASEOF      ( 0 addr1 addr2 ...
                   addrn 4 -> .
                   addrx is the addr of
                   an inline offset
                   following a BRANCH
                   compiled by an ESAC.
                   Executes ?CASE
                   syntax check; 0 on
                   the stack is an
                   end-of-data signal
                   which was pushed by
                   CASEOF;  For each
                   CASE...ESAC, patches
                   the offset at addrx
                   so that the BRANCH
                   compiled by ESAC
                   will branch to the
                   R>DROP which END-
                   CASEOF  compiles.)
                   4 ?CASE ( Syntax check)

BEGIN -DUP WHILE           ( there's a nonzero offset on stack)
  2                        ( ENDIF will be checking for this)
  [COMPILE] ENDIF          ( ENDIF will compute, emplace offset)
                         REPEAT
COMPILE R>DROP ;           ( code drops case value from R stack)
IMMEDIATE

ALIAS SELECTOR   R

( PRODS/PROD/DCRP/CATCHAL/ENDPRODS are analogous to
  CASEOF/CASE/ESAC/OTHERWISE/ENDCASEOF except there
  is no selector value:  each PROD tests for tf on stack.)

33 CONSTANT      PROSYNTX ( Error number, production set syntax)

: PRODS          ( -> 0 6  .   Compile-time setup for PROD set.)
                 0         ( end-of-data signal to ENDPRODS)
                 6 ;       ( For PROD syntax check)

IMMEDIATE

: ?PROD          ( n1 n2 -> . )
                 <> IF PROSYNTX ERROR ENDIF ;

: PROD           ( 6 -> addr 7 )
                 6 ?PROD

                 6 ;
IMMEDIATE

: ENDPRODS       ( 0 addr1 addr2 ... addrn 6 -> )
                 6 ?PROD                ( Syntax check.)
                 BEGIN -DUP WHILE       ( there's a nonzero offset of stack)
                 2                      ( ENDIF will be checking for this)
                 [COMPILE] ENDIF        ( ENDIF will compute, emplace offset)
                 REPEAT ;
IMMEDIATE
```

Steve Brecher
Software Supply
Long Beach, CA

Judges' Comments -

This entry supports essentially the same syntax and semantics as the FORTH-85 CASE statement (see FD I/5), but offers the following advantages:

1. Compile-time syntax checking.
2. Explicit OTHERWISE clause.
3. Case selector is kept on return stack instead of in a special variable. This allows nesting of CASE constructs.
4. 16-bit branch offsets are used, rather than a mixture of 16-bit addresses and 8-bit offsets. This eliminates the need for a special run-time END-CASE word and simplifies compilation.

NEW PRODUCT

Z-80

We have a Z-80 implementation of FIG-FORTH that was derived directly from 8080 FIG-FORTH 1.1 and will run under either CP/M or Cromemco CDOS. The code is optomized to exploit the additional Z-80 registers and instructions. Although this was developed for our own internal use we are willing to make it available at cost to interested FIG members. For $25.00 to cover media, copying, and shipping, we will send two soft- sectored single density eight inch diskettes containing executable Z-80 FORTH interpreter, all source files, and sample FORTH programs. Payment may be sent by check or money order to the address below. Please allow us 30 days for shipment. LABORATORY MICROSYSTEMS, 4147 Beethoven Street, Los Angeles, CA 90066, (213) 390-9292.

A CASE STATEMENT

Mike Brothers

Approximately a year ago I was writing a program and needed a more powerful branching construction than the standard IF..ELSE..ENDIF construction. Somehow I decided on implementing Pascal's CASE statement in FORTH, and this is the one which is described here. This CASE statement is also included in the standard SL5 package, available from the Stackworks.

Some of the advantages of SL5's CASE statement are:

1) Infinite nesting is possible.

2) The CODE is machine independent.

3) Programs are easier to read because of its simplicity.

CASE statement definitions

```
: $CASE R> DUP 2 + SWAP @ >R >R ;
: $=: OVER = IF
     DROP R> 2 + >R
     ELSE R> @ >R
  ENDIF ;
: $;; R> DROP ;              : NOCASE DUP ;
: CASE  \ $CASE   HERE 0 , ;                        IMMEDIATE
: =:  \ $=:   HERE 0 , ;                            IMMEDIATE
: ;;  \ $;;    HERE SWAP ! ;                        IMMEDIATE
: CASEND  \ R>  \ 2DROP   HERE SWAP ! ;             IMMEDIATE
```

Compilation

During compilation, "CASE" compiles the address of "$CASE" and a 0 for the address field. Every subsequent "=:" causes "$=:" to be compiled along with a dummy address field (to be set by the next ";;"). The word ";;" then compiles "$;;" and replaces the address field of the previous "=:" with addrx. When "CASEND" is finally processed, the something resembling figure 2 should be present.

```
              -------     --------
             | $CASE |   | ADDR1  |
              -------     --------
                          -----    -------                     -----
 addrA:  ExpressionA1    | $=: |  | addrB |   expressionA2    | $;; |
                          -----    -------                     -----
                          -----    -------                     -----
 addrB:  ExpressionB1    | $=: |  | addrC |   expressionB2    | $;; |
                          -----    -------                     -----
                               .
                               .
                               .
                          -----    -------                     -----
 addrY:  ExpressionY1    | $=: |  | addrZ |   expressionY2    | $;; |
                          -----    -------                     -----
             ----      -------
 addrZ:  |  R>  |    | 2DROP |
             ----      -------
 addr1:
```

Execution

Upon entry, a number corresponding to the case is assumed to be on the stack. "$CASE" then places ADDR1 on the return stack. ExpressionA1 is then executed, and "$=:" compares the value on the tos (top of stack) with the nos (next on stack), which should be the entry value. If these are equal, the entry value is dropped and ExpressionA2 is executed before "$;;" sets the interpreter pointer to ADDR1 (which is on the return stack). If the two values are not equal, "$=:" sets the IP to addrB and execution continues until a valid case is found or the cases are exhausted, which causes ADDR1 to be removed from the return stack and the entry value dropped.

The word "NOCASE" always causes the $=: to execute the following expression, simply by setting the tos equal to the entry value.

Examples of CASE statement usage

```
: EXAMPLE1 BEGIN
    ." CHOICE? "  GCH CASE
       41 =: ." APPLE "       1 ;;     ( A is for APPLE )
       42 =: ." BLUEBERRY "   1 ;;     ( B is for BLUEBERRY )
       43 =: ." CHERRY "      1 ;;     ( C is for CHERRY )
       44 =: ." DATE "        1 ;;     ( D is for DATE )
       45 =: ." ELDERBERRY "  1 ;;     ( E is for ELDERBERRY )
       NOCASE =: ." WRONG "   0 ;;     ( repeat till valid )
    CASEND
 END ;
```

Figure 3. Example of CASE statement usage

CASE statement example

The example shown above illustrates the CASE statement's simplicity and power. When EXAMPLE1 is executed, a character is read from the keyboard. If the character is an "A", the string "APPLE" is displayed. If the character is a "B", the string "BLUEBERRY" is shown. If none of the five are selected, the string "WRONG" is displayed and the loop is executed again until a valid (A-E) choice is entered.

The COND Statement

One particular advantage of the case statement is that an additional branching structure which executed an expression based on a boolean expression can be defined with a few more words. I call this structure the COND statement, and the extra words needed are shown in figure 3. The structure is much like that of the CASE statement, as shown in the example in figure 4.

```
: COND  COMPILE CASE ;                                  IMMEDIATE
: CONDEND  \ R>    \ 2DROP    HERE SWAP ! ;             IMMEDIATE
: $:: IF
     R> 2 + >R
     ELSE R> @ >R
 ENDIF ;
: ::  \ $::     HERE 0 ,  ;                             IMMEDIATE
```

Figure 4. COND statement definitions

```
: EXAMPLE3 COND
      DUP 0>  ::  T" POSITIVE"  ;;
      DUP 0=  ::  T" ZERO" ;;
      1       ::  T" NEGATIVE"  ;;
 CONDEND ;
```

Figure 5. Examples of COND statement usage

The COND Example

The Example shown above illustrates the similarity between the COND construction and the CASE statement. Upon entry to EXAMPLE2, an integer is assumed to be on the stack. One of the strings "POSITIVE", "ZERO", or "NEGATIVE" is displayed depending on the integer.

Mike Brothers
The Stackworks
Bloomington, IN 47401

Judges' Comments - This is a practical method but not as portable as it might appear. The 2+ in $CASE and $=: will have to be relocated for preincrementing 6800 systems. The COND statement is a nice variation on CASE.

NEW PRODUCT

FORTH FOR CP/M

Mitchell E. Timin Engineering Co. has an enhanced version of FIG FORTH ready for immediate delivery. It is supplied on an 8 in. single density diskette, ready to run on any system with CP/M and at least 24K of memory. A FORTH style editor with 20 commands is included, as well as a virtual memory sub-system for software which is permanently stored on diskettes, then loaded when needed. The user may also make permanent additions to the resident FORTH vocabulary. A Z-80/8080 assembler is also included, allowing the user to create new FORTH definitions which compile directly into machine code. All Z-80 or 8080 instructions may be used. The IF...ELSE..., BEGIN...UNTILL, and BEGIN...WHILE...control structures may be included in assembler definitions; these will automatically compile into appropriate machine code.

Other enhancements include an interleaved disk format that minimizes the time required for disk access. A 1024 byte disk block may be read or written in as little as 1/6 second. Eight of these blocks are maintained in RAM for immediate access and automatically swapped with others on the disk as they are needed.

The price is $75 for the 8 in. single density version, $90 for other diskette formats. Adequate documentation is included, suitable for the beginner as well as the experienced computer user.

FIG FORTH was originally defined by the FORTH INTEREST GROUP and is very close to the FORTH-79 international standard.

Mitchel E. Timin Engineering Co., 9575 Genesse Avenue, Suite E-2, San Diego, CA 92121.

DO-CASE STATEMENT

Dwight K. Elvey

OVERVIEW OF STATEMENT:

This is a DO-CASE written in FIG FORTH. It allows the operations of statements on the condition of a match of a case value and a case key. This DO-CASE also has a range case that allows the use of the condition to be done on a range of case key values. The NOT CASE and the NULL CASE concept are also allowed in this DO-CASE.

```
SCR # 19
( DO-CASE ALSO COMPILE { ...} LIKE COMPILE )
: } ;
: COMPILE{ ?COMP BEGIN R> DUP 2+ >R @ DUP ' }CFA = IF DROP
1 ELSE , 0 ENDIF UNTIL ;

: DO-CASE COMPILE >R 0 5 ; IMMEDIATE
: CASE 5 ?PAIRS COMPILE{ R = 0BRANCH } HERE 0 , 7 ; IMMEDIATE
: RANGE-CASE 5 ?PAIRS COMPILE{ R SWAP - 0< 0= 0BRANCH } HERE 0 ,
 COMPILE{R - 0< 0BRANCH} HERE 0 , HERE COMPILE BRANCH HERE 0 ,
 HERE SWAP >R ROT >R R - R> ! OVER - SWAP ! R> 7 ; IMMEDIATE
: END-CASE 7 ?PAIRS COMPILE BRANCH HERE 0 , SWAP HERE OVER -
 SWAP ! SWAP 1+ 5 ; IMMEDIATE
: END-DO-CASE 5 ?PAIRS -DUP IF 0 DO HERE OVER - SWAP ! LOOP
 ENDIF COMPILE{ R> DROP } ; IMMEDIATE ;S
```

```
SCR # 29
  ( EXAMPLE OF DO CASE )
: EXAMPLE DO-CASE
4 CASE ." THE NUMBER WAS 4 " CR END-CASE
5 3 RANGE-CASE ." THE NUMBER IS 3 OR 5 " CR END-CASE
6 CASE ." THE NUMBER IS 6 " CR END-CASE
( NULL OR NOT CASE ) ." THE NUMBER ISN'T 3,4,5 OR 6 " CR
END-CASES ; ;S
```

COME TO FIG CONVENTION NOVEMBER 29

WHAT EACH DEFINITION FOR DO-CASE DOES:

DO-CASE consumes the case key value to be used later by the individual cases. This is the initialization statement for a DO-CASE field.

CASE does a comparison of the case key value and a case value. If a match is found the statements between CASE and the next END-CASE are done, then operation is picked up after the END-DO-CASE statement; else operation continues after the END-CASE statement and continues until END-DO-CASE or the next successful case.

RANGE-CASE does a comparison of the case key value and an inclusive range of values set by the two case values. The first case value on the stack must be greater in value then the next case value on the stack. The operation of RANGE-CASE is otherwise the same as CASE.

END-CASE indicates that the conditional CASE or RANGE-CASE is ended. It must be paired with any use of CASE or RANGE-CASE.

END-DO-CASE is used to close a DO-CASE field. Its main purpose is to do the cleaning of the stack and provide an exit point for the CASE statements. DO-CASE must be paired with a closing END-DO-CASE.

GLOSSARY ENTRIES

CASE

n ---   (run-time)
n --- addr n   (compile)

Used in a colon-definition in the form:

```
n(1) DO-CASE ... n(2) CASE (tp) ... END-CASE
```

(fp) ... END-DO-CASE

At run-time a comparison of n(1) and n(2) is done. If there is a match the true part is executed, then execution resumes after END-DO-CASE. If there is no match execution continues at the false part (fp). It must be followed by an END-CASE and an END-DO-CASE. It must be preceded by a DO-CASE.

At compile-time CASE compiles a branch and reserves space for an offset at addr. addr and n are used by END-CASE to resolve the offset and for error testing.

DO-CASE

n --- (run-time)
--- nl n2 (compile)

Used in a colon-definition in the form:

n(1) DO-CASE ... n(2) CASE (tp) ... END-CASE

(fp) ... END-DO-CASE

At run-time it consumes the value on the stack to be used later by case statements. This is used to initialize a do case field. See CASE for its use.

At compile-time DO-CASE leaves a case count (nl) and a value for error testing (n2).

END-CASE

--- (run-time)

nl addrl n2 --- addr2 n3 n4 (compile)

At run-time it is used to terminate a CASE or RANGE-CASE statement. See CASE or RANGE-CASE for its use.

At compile-time it takes a value for an error check (nl), an address (addrl) to resolve an offset and a value that is the number of cases. It leaves a value for error checking (n4) , a value with a new case count (n3 = nl + 1) and an offset at address (addr2) to be used later.

END-DO-CASE

--- (run-time)
addr(1) addr(2) ... addr(nl) nl n2 --- (compile)

At run-time this terminates a DO-CASE field. See DO-CASE or CASE for its use.

At compile time it takes a case count (nl) and the count number of addresses to be used to resolve offsets and a value to use for error checking (n2).

RANGE-CASE

nl n2 --- (run-time)
n --- addr n (compile)

Used in a colon-definition in the form:

n(1) DO-CASE ... n(2) n(3) RANGE-CASE (tp)
... END-CASE (fp) ... END-DO-CASE

At run-time a comparison of n(1) and the inclusive range of n(2) and n(3) is done. If there is a match the true part (tp) is executed, then execution resumes after END-DO-CASE. It must be preceded by a DO-CASE. n(2) must be greater than or equal to n(3) to do a successful case.

At compile-time RANGE-CASE compiles a branch and reserves space for an offset at addr. addr and n are used by END-CASE to resolve the offset and for error testing.

EXAMPLE OF USE:

SCR # 29 is an example of the use of DO-CASE. It shows the use of CASE, RANGE-CASE and null or not-case. In order to use it type in SCR # 19 first then SCR # 29. It is used by typing a number, then EXAMPLE. The result will be a comparison of the number you typed and the comparisons done in the DO-CASE.

ADVANTAGES AND DISADVANTAGES:

The main disadvantage is that DO-CASE uses the return stack like DO ... LOOP does. This means that a value can not be passed on the R-stack from the outside of the DO-CASE field to the inside or vice-versa. Also this means that if the loop value I is to be used it must be on the operation stack before entering the DO-CASE.

The advantages of this DO-CASE are that it has a RANGE-CASE and the ability to allow the concept of not or null-case. This allows it to be used for something like an input entering routine for something like an editor. The CASEs can be used to prescan for special keys, the RANGE-CASEs can be used as a capitals only routine and the null-case used to do the normal entry.

Dwight K. Elvey
Santa Cruz, CA 94065

Judge's Comments -

This entry performs the functions of the FORTH-85 CASE statement. It also provides compile-time syntax checking, allows a range of indices to be treated as a single case, and offers a "none-of-the-above" case.

Compiling the same list of run-time words for each case results in excessive space overhead (about 28 bytes for each RANGE-CASE). Defining some new run-time words would save space without adding much execution time.

Also, using " - 0<" to check the index against a range gives the wrong result if the subtraction overflows.

FIG NORTHERN CALIFORNIA MONTHLY MEETING REPORT

28 June 80

FORML Session -

Tom Zimmer described the product of his last two weeks effort - tinyPASCAL (written in FORTH, of course). Two of his remarkable routines include the use of Ragsdale's table structure (C.F., Morse code tutorial, 24 May 80 FIG Meeting) in a Tokenizer and his technique of recursion uning dummy pointer-variables. The PASCAL design came from a "Byte" (Sep-Nov 78) series of articles which instructed the reader to do it in BASIC. Tom's first version of PASCAL-under-FORTH occupies some forty blocks.

FIG Meeting -

Three technical talks were delivered. Michael Perry described a CP/M File System written in FORTH which gives a 8080/Z80 version of FORTH compatibility with and use of extant CP/M data files.

Kim Harris spoke about arrays, i.e. how tables are created by alloting space to named variables and accessing array components by manipulating an index.

Bill Ragsdale discussed database concepts after FORTH, Inc.'s poly-FORTH and the organization of fields within files. Their FORTH definitions and demonstrations of file manipulation "How to talk to mass storage".

Announced was the availability of source code for FORTH on the 6809 running under SWTPC's FLEX 1.9. This is copyrighted by Talbot and is available from FIG for $10. See order blank.

Regarding FIG organizational business, two volunteers were asked to step forward - one to organize meetings, sequence schedules and distribute tasks (Ragsdale estimates 3 hrs/mo effort needed) and the other to take up the meeting announcement effort.

;s Jay Melvin

=A CASE IMPLEMENTATION=

William S. Emery

Yet another CASE implementation, this for either the TI990 or the Motorola 6800.

The objectives of this implementation were:

1. To provide a clear source program structure when using CASE, i.e. no compiler directives.
2. To provide a direct exit from any executed CASE to the next program statement.
3. To provide an ELSE (or Trap) statement within the CASE structure.

Please note: in both my 990 and 6800 implementations of FORTH all compiled addresses are 16 bits. No relative addressing is used.

The compiling word 'CASE creates a dictionary entry as follows:

1. The code address of (CASE).
2. The source argument to be compared. This eliminates the compilation of LITERAL and the necessity of moving the argument to the stack.
3. The branch address for I when not true. This is the address of the next CASE statement in the list.

A complete CASE statement requires three unique words:

<CASE , pronounced "open case,"
CASE , pronounced "case," and
CASE> , pronounced "case closed."

A sample of use is:

```
: TEST
      <CASE 1  ." FIRST"
       CASE 5  ." FIFTH"
       CASE 7  ." SEVENTH"
       ELSE .  ." NOT VALID"
       CASE> ;
```

At compile time <CASE places a zero delimeter on the stack, compiles to the dictionary (CASE), the source argument, and a nul, which will become the not true branch address. CASE then compiles a standard ELSE, which resolves the preceding not true, and deposits a nul address, to be resolved by CASE>. The address of this nul cell is left on the stack. Finally, CASE> resolves all addresses on the stack to itself until the opening nul is encountered.

```
CODE (CASE)       ( TI990 ASSEMBLER )
     I )+ S ) C      ( COMPARE ARGUMENT TO STACK )
     0= IF    S INCT    I INCT  ( POP STACK ENTER PRCC )
        ELSE    I ) I MOV       ( SET UP BRANCH ADDR )
        THEN    NEXT

: 'ELSE    \  (ELSE)   HERE 0 ,    HERE ROT !   ;
: #,          32 WORD  NUMBER , ;

: 'CASE    \ (CASE)    #,    HERE 0 ,   ;

: <CASE       0    'CASE  ;                          IMMEDIATE
: CASE       'ELSE 'CASE  ;                          IMMEDIATE
: CASE>      BEGIN   HERE SWAP !    ?DUP 0= END  ;   IMMEDIATE
```

A dictionary map of the compiled source would be as follows:

```
(headers omitted - addresses in hex )
XX00    (case) 0001 XX12 (.") 5F IR ST (else) XX44
XX12    (case) 0005 XX24 (.") 5F IF TH (else) XX44
XX24    (case) 0007 XX34 (.") 7S EV EN TH
XX34    (else) XX44 (.") 9N OT bV AL ID
XX44    (;)
```

While using byte offset addressing for the branches would have saved one or two bytes per CASE statement, to do so would violate the definition of word aligned dictionary established at the recent Standards Team meeting.

The word incorporating the CASE paragraph is entered with any 16 bit value on the stack. Any CASE statement finding the stack equal to its argument pulls the entry from the stack. If no CASE statement matches the stack parameter the value remains for the ELSE statement, if used, or beyond the "case closed" point.

This procedure executes (and compiles) nicely on the byte oriented Motorola 6800 by using the following definition for (CASE).

```
CODE (CASE)       ( M6800 ASSEMBLER )
     I LDX   0 ) LDX  N STX    ( SAVE ARGUMENT )
       TSX   0 ) LDX  N CPX    ( COMPARE TO STACK )
     0= IF     A PUL  B PUL    ( POP STACK )
               I LDX  INX INX INX INX  ( ENTER PROC )
        ELSE   I LDX  2 ) LDX  I  STX  ( SET BRANCH )
        THEN   NEXT
```

Thank you for the opportunity to submit this. I think the contest idea is a great one. How about some future contests on +LOOP, the Bartholdi "TO" concept and/or Data Structures. If publication space permits I'd also be interested in a competition on SORT and/or an approach to precompiled, relocatable FORTH for virtual memory processing.

William S. Emery
Costa Mesa, CA 92626

Judges' Comments - This entry achieves its objectives with only 7 short and well-factored new word definitions. The CODE word could have been written in high-level. While having to specify the case keys as numbers at compile time is a restriction, it is adequate for many applications. And it does simplify the source code.

FIG NORTHERN CALIFORNIA MONTHLY MEETING REPORT

26 July 80

FORML Session -

Henry Laxon presented his string package which has been his first FORTH programming effort. He pointed out that this package was designed for a computerized type-setting task and not text editing. The word "string" takes a length parameter and name and is manipulated so to find, concatenate, parse, move and so forth.

John Cassady then outlined his string package which he fashioned after Northstar's BASIC. He pointed out it's file handling utility and a discussion arose regarding screen windows, input windows and video segmentation. Amazing how FORTH gets strung along.

FIG Meeting -

Announcements included the report of over 25 attendees at Kim Harris' Humbolt State FORTH class.

Allyn Saroyan described the problems he's had trying to convert code from other machines and asserted that we ought to submit code along with its algorithm and perhaps even assembler particulars.

Don Colburn, from Creative Solutions, mentioned a FORTHcoming tutorial under CP/M with stackgraphics.

Bob Smith reviewed progress and problems of the floating point standards team effort.

John James described Cap'n Software's Apple editor.

Bill Ragsdale spoke briefly about the Installation Manual version editor and code was shown on how to extend FORTH, Inc.'s editor.

A preview copy of the August 1980 Byte magazine was passed around. See the order form to get your copy.

;s Jay Melvin

APPLE - 4th CASE

F.W. Fittery

Here is a select case for Apple-4th. The Apple works so-far and allows any level of nesting of any of the allowable structures plus more BEGIN-CASES, END-CASES. You will get a lot of failures if you do not balance your (BEGIN-CASES==END-CASES) and your (CASE==END-CASE). Also be aware the the top of stack is still available if none of the case statements are executed. Otherwise the top of stack is eaten up by the case statement. When BEGIN-CASES is encountered 0 is placed on the stack for END-CASES. When CASE is encountered at compile time OVER = ZERO-BRANCH 0 , DROP is compiled inline. When END-CASE is encountered the ZERO-BRANCH for the matching case is patched to the proper jump point. When END-CASE is found all forward jumps set-up by END-CASE are resolved. This is done with a BEGIN END looking for the 0 put on the compile time stack by BEGIN-CASES. Good luck.

Note: The general approach of the CASE statement is:

```
:TEST 5 OVER = IF DROP ." FIVE "  ELSE
      6 OVER = IF DROP ." SIX  "  ELSE
      7 OVER = IF DROP ." SEVEN"  ELSE
      DROP ." BAD INPUT"
               THEN THEN THEN ;
```

Generates the same code as:

```
: TEST
    BEGIN-CASES
      34 CASE 34 . END-CASE
      35 CASE 35 . END-CASE
      36 CASE 36 . END-CASE
      DROP ." BAD INPUT"
    END-CASES ;
```

Note: You must use up so if no case is executed as if is left on the stack.

Case Documentation

The CASE statement format is as follows:

The result of the BEGIN-CASES, END-CASES is:

1 0 if a CASE option is executed
1 1 if no CASE option is executed

If no CASE option is executed the flow of execution starts after the last END-CASE. Because of this and the fact that the top of stack passed to the BEGIN-CASE is still on top of the stack you may drop the parameter or you may use it to do a calculation which is done only when none of the case options are selected:

Note: Though the code executes exactly the same code the format in Figure 1 is much easier to understand than that in Figure 2. It is also much preferable.

Case Statement

Figure 1.

```
: ESC-ESC ." ESC-ESC" ;
: NEC ." ESC-CTL-N" ;
: LEC ." ESC-CTL-L" ;
: SEC ." ESC-CTL-S" ;
: ESC KEY
      BEGIN-CASES
        27 CASE ESC-ESC END-CASE
        14 CASE NEC     END-CASE
        12 CASE LEC     END-CASE
        19 CASE SEC     END-CASE

      END-CASES ;
: OUTPUT
    BEGIN-CASES
      27 CASE ESC     END-CASE
      14 CASE 91 DOT  END-CASE
      12 CASE 92 DOT  END-CASE
      19 CASE 95 DOT  END-CASE
      DOT
      END-CASES ;
```

```
 : MONITER BEGIN KEY DUP OUTPUT
          32 = END ;
;S
```

Figure 2.

```
: ESC-ESC ." ESC-ESC" ;
: SDC ." ESC-CTL-S" ;
: LEC ." ESC-CTL-L" ;
: NEC ." ESC-CTL-N" ;
: OUTPUT
    BEGIN-CASES
      27 CASE KEY
           BEGIN-CASES
              27 CASE ESC-ESC END-CASE
              14 CASE NEC     END-CASE
              12 CASE LEC     END-CASE
              19 CASE SEC     END-CASE
              .
           END-CASES
                      END-CASE
      14 CASE 91 DOT  END-CASE
      12 CASE 92 DOT  END-CASE
      19 CASE 95 DOT  END-CASE
      DOT
      END-CASES ;
 : MONITER BEGIN KEY DUP OUTPUT
          32 = END ;
;S
```

Support Words

```
: DOT 0  'S 1 + 1 TYPE DROP ;
: '?  WORD HERE CONTEXT @ @ FIND ;
: LIT R> 2 + DUP >R @ 2 + ;
: .PFETCH R> R> 2 + DUP >R @ 2 + SWAP
         >R ;
;S
```

(CASE)

```
: BACKSLASH .PFETCH 2 - , ;
: BRANCH R> 2 + @ >R ;
: ZERO-BRANCH 0= IF R> 2 + @ >R
                 ELSE R> 2 + >R THEN ;
: BEGIN-CASES 0 ; IMMEDIATE
: OVER= OVER = ;
: CASE BACKSLASH OVER=
  BACKSLASH ZERO-BRANCH HERE 0 ,
  BACKSLASH DROP ; IMMEDIATE
: END-CASE BACKSLASH BRANCH HERE 0 ,
  SWAP HERE 2 - SWAP ! ; IMMEDIATE
: END-CASES BEGIN -DUP IF HERE 2 - SWAP
  ! 0 ELSE 1 THEN END ; IMMEDIATE
PRINT-OFF
85 85 PLIST
( GENERAL 8-BIT SELECT CASE CODE        )
( NEEDS BACKSLASH ( \ )  TO WORK        )
( FOR 16 BIT VERSION SEE # S 1,2,3      )
: XXX IF ELSE THEN ;
' XXX DUP @ SWAP 2 + @
( #1:  2 BECOMES 1 IN THE ABOVE LINE)
  FORGET XXX
  CONSTANT BRANCH
  CONSTANT ZERO-BRANCH
: BEGIN-CASES 0 ; IMMEDIATE
: OVER= OVER = ;
: CASE BACKSLASH OVER=
  ZERO-BRANCH , HERE 0 ,
  BACKSLASH DROP ; IMMEDIATE
: END-CASE BRANCH , HERE 0 ,
  SWAP HERE 2 - SWAP ! ; IMMEDIATE
( #2:     2 BECOMES 1 IN THE ABOVE LINE)
: END-CASES BEGIN -DUP IF HERE 2 - SWAP
( #3:     2 BECOMES 1 IN THE ABOVE LINE)
  ! 0 ELSE 1 THEN END ; IMMEDIATE
```

E. W. Fittery
International Computers
Mount Arlington, NJ 07856

Judges' Comments - Interesting but rather limited.

COME TO FIG CONVENTION
NOVEMBER 29

DO-CASE EXTENSIONS

Bob Giles

Upon using the DO-CASE structure offered by Rick Main in the Vol. 1, No. 5 issue of Forth Dimensions, I came across several instances where the power of this tremendously useful construct can be improved. The first is where several options are defined using the CASE and END-CASE structure, but all remaining cases have a common option. The other feature is where the DO-CASE variable is to be tested within a certain range of values instead of strict equality to one value per CASE. In order to maintain symmetry, some renaming of the keywords was necessary. The old structure looks like this:

```
DO-CASE
  w CASE.............END-CASE
  x CASE.............END-CASE
               ...
               ...
  z CASE.............END-CASE
END-CASES
```

My structure looks like this:

```
DO-CASE
  a CASE.............END-CASE
b c CASES...........END-CASES
               ...
               ...
  J CASE.............END-CASE
k l CASES...........END-CASES
  m CASE.............END-CASE
    OTHERWISE........
END-DO-CASE
```

The lower case letters indicate operations that leave a 16 bit value on the stack. DO-CASE is symmetrical with END-DO-CASE, CASE is symmetrical with END-CASE, CASES is symmetrical with END-CASES and OTHERWISE, well....

OTHERWISE is useful when there are several courses of action for certain values of the DO-CASE variable, and a common routine for all the other cases. This closes any "loopholes" for erroneous values that can occur. This is easily implemented by putting the common routine after the last END-CASE and before the END-CASES in Rick's DO-CASE structure. However, for readability and documentation, I defined a dummy word, OTHERWISE , (i.e. : OTHERWISE ; IMMEDIATE), to mark the action. Making this work an IMMEDIATE word assures that run time is not affected. OTHERWISE must be used at this particular point in the DO-CASE structure, and has no meaning or usage anywhere else.

The need to test for equality to a value within a range leads to the CASES structure. Whereas x CASE tests the DO-CASE variable (VCASE) for x = VCASE, lo hi CASES tests VCASE to see if it satisfies lo ≤ VCASE ≤ hi. If VCASE is within the range of the lower boundary, lo, and the higher boundary, hi, then the appropriate statements are executed within the CASES...END-CASES statement (this is the newer word - don't confuse it with Rick's END-CASES). If VCASE is out of range, these statements are skipped and execution resumes after the END-CASES (new word) statement.

The listing of the structure is in the figure (see enclosure). The minor changes include - changing the name of END-CASES, making a dummy word called OTHERWISE, and defining the new word CASES.

The simplicity of CASES does not reflect the time it took to get it working. (A fairly lengthy interactive Forth debugger was written to help with the development). The basic idea is to subtract the upper limit from VCASE minus one and see if the result is zero or positive (i.e., the carry flag IS set). If the carry flag IS set, then the result is out of range and the Forth instruction pointer (kept in the BC pair) has to be incremented so that the next "instruction" executed will be the one after END-CASES. The action is the same as when VCASE does not match in the CASE statement. If the carry flag is NOT set, then VCASE is less than or equal to the upper bound and possibly in range. If VCASE is less than the upper bound, the lower bound is subtracted from VCASE. If the result is negative (i.e., carry is NOT set), then VCASE is out of range and IP is incremented to resume after END-CASES. If the result is positive or zero (i.e., the carry flag IS set), then VCASE is between or equal to the upper and lower boundaries. In this case, the statements between CASES and END-CASES are executed. At END-CASES, execution jumps to after the END-DO-CASE statement and continues.

Two interesting concepts were included in this implementation. The first was the use of the assembly language CALL. The ' (tick) causes the code field pointer of the next word to be placed on the stack. The Forth CALL takes this address from the stack and assembles the CALL opcode and the address into the dictionary. At run time, the call to the -TOP subroutine is executed, and the 8080 program counter is pushed on the top of the stack. Within -TOP, the H POP takes the return address to HL, and then exchanges it with the top item (the boundary) so that the return address will be on top of the stack when RETurn is executed.

' AFTER-END-CASES leaves an address on the top of the compile time stack which is assembled into the dictionary by the JMP in code CASES. At run time, this AFTER-END-CASES segment serves as an extension to machine code in code CASES. Although this type of programming is a GOTO type of construct, it is used here to keep the definition of code CASES short. It also adds insight as to the intent of extended segment by the use of a name. My advice to other programmers is to use this jump around feature very sparingly, so as to remain in keeping with the concepts of structured programming.

The TEST for the new DO-CASE is listed on screen 153. It differs from the program that Rick submitted in that the various variables are to be entered on the stack before executing TEST. This way, all 65,536 possibilities can be tried instead of only the 128 available from an ASCII keyboard.

All of the following was done using Zendex SBC-FORTH V 1.0 for an 8080 processor.

A final note is in order. The earlier DO-CASE had a bug in it pertaining to the address used to store VCASE. Notice that my routines deleted the ' (tick) which preceded VCASE in lines 3 and 4 of the first screen that Rick sent (see Vol. 1, #5 of Forth Dimensions, pg. 51). This is because ' VCASE causes the address of the parameter field to be put on the stack, rather than the location of VCASE in the RAM area. Although the earlier DO-CASE works, fetching VCASE always yields a zero.

Bob Giles
Magnetic Media, Inc.
Tulsa, OK

Judges' Comments -

More of an extension of previous work than a new CASE.

```
SCREEN  150
( DO-CASE STATEMENTS                        4-4-80 BG )
BASE C@
  VOCABULARY FORTH+    FORTH+  DEFINITIONS

  ( DO-CASE CASE END-CASE CODE DEFINITIONS )
0 VARIABLE VCASE
CODE DO-CASE H POP VCASE SHLD I INX I INX NEXT JMP
CODE CASE W POP VCASE LHLD L A MOV W 1+ CMP
     0= NOT IF I LDAX I 1+ ADD A I 1+ MOV NEXT JNC
               I INR NEXT JMP THEN H A MOV W CMP
     0= NOT IF I LDAX I 1+ ADD A I 1+ MOV NEXT JNC
               I INR NEXT JMP THEN I INX NEXT JMP
CODE END-CASE  I LDAX A L MOV I INX I LDAX A H MOV
     H PUSH I POP NEXT JMP
BASE C! ;S    ( END CODE DEFINITIONS )
( COPIES FROM FORTH DIMENSIONS V1-5 pg  50/51    BG 4-4-80 )


SCREEN  151
( DO-CASE EXTENTIONS                      6-2-80 BG   )

CODE -TOP     H POP  XTHL  XCHG  E A MOV  CMA  A E MOV  D A MOV
              CMA  A D MOV  D INX  D DAD RET
(NOT TO BE CALLED FROM HIGH-LEVEL)
CODE AFTER-END-CASES  B LDAX  C ADD  A C MOV  NEXT JNC
                      B INR   NEXT JMP

CODE CASES  VCASE LHLD H DCX XCHG ' -TOP CALL
    CS      IF D POP ' AFTER-END-CASES JMP  THEN
            VCASE LHLD XCHG ' -TOP CALL
    CS NOT  IF ' AFTER-END-CASES JMP  THEN  B INX NEXT JMP

CODE END-CASES  I LDAX  A L MOV  I INX  I LDAX  A H MOV  H PUSH
                I POP   NEXT JMP
    ;S


SCREEN 152
( CASES&OTHERWISE EXTENSIONS            5-22-80 BG )
( FORTH+  DEFINITIONS - COMPILER DO-CASE STATEMENTS )

: DO-CASE  COMPILE  DO-CASE HERE 0 0 , ; IMMEDIATE
: CASE     COMPILE  CASE SWAP HERE 0 C, ; IMMEDIATE
: END CASE COMPILE  END-CASE HERE 0 , SWAP HERE
           OVER - SWAP C! ; IMMEDIATE
( COPIED FROM FORTH DIMENSIONS V1-5 pg  50/51    BG 4-4-80 )

: CASES  COMPILE CASES SWAP HERE 0 C, ; IMMEDIATE
: END-CASES  COMPILE  END-CASES HERE 0 , SWAP HERE OVER - SWAP
            C! ;   IMMEDIATE
: OTHERWISE ; IMMEDIATE   ( NULL DEFINITION )
: END-DO-CASE BEGIN HERE SWAP ! -DUP 0 = END ; IMMEDIATE
FORTH+     ;S



SCREEN  153
( TEST FOR EXTENDED DO-CASE                  5-22-80 BG )
 BASE C@ HEX
: MONITOR     DO-CASE
          40  CASE  QUIT         END-CASE
          41  CASE  ." AAAA "    END-CASE
          42  CASE  ." BBBB "    END-CASE
          43  CASE  ." CAT  "    END-CASE
      30  39  CASES ." NUMBERS " END-CASES
      OFE 102 CASES ." CROSS "   END-CASES
              OTHERWISE ." NOT TESTED "
              END-DO-CASE ;

: TEST  BEGIN DUP MONITOR 0 = END ;

BASE C! ;S

```

ENTRY FOR THE FIG CASE CONTEST

Arie Kattenberg

An Overview of the CASE Statement

Externally the CASE statement looks like:

```
m  n  CASE ........... ESAC
   k  CASE ........... ESAC
   .  . .                . .
   .  . .                . .
   .  . .                . .
   .  . .                . .
   .  . .                . .
   .  . .                . .
   l  CASE ........... ESAC
      ENDCASE
```

- If a comparison is not 'true' (m≠n) the m stays on the stack and is tested against the next CASE.
- If a CASE is met the m is dropped and after the case body is executed, the ESAC transfers control to words following ENDCASE.
- If none of the CASES is met, ENDCASE has compiled a DROP that now drops the m instead of one of the CASES doing that.

If we want explicitly some (stack) operations to be done when none of the cases is met, the m that remains on the stack there would be bothering. We then use:

```
m  n  CASE ........... ESAC
   k  CASE ........... ESAC
   .  . .                . .
   .  . .                . .
   .  . .                . .
   .  . .                . .
   l  CASE ........... ESAC
      OTHER .......... ENDCASE
```

Now the 'OTHER' has compiled a drop for the m and ENDCASE does not compile a drop.

In both the above examples we can nest other case structures in any of the case bodies. This is another reason for using 'OTHER' sometimes.

Though this is in no way essential to the above structures I have chosen a high level branch in the conditional branch that is compiled by CASE (i.e. (CASE) manipulates the return stack contents to effectuate a branch). Now it is simple, machine independent and self explaining to make words like:

```
>CASE  <CASE  CASES  ODDCASE  etc.
```

that can take the place of CASE in the above examples. (Of course this can be done using machine language conditional branches for these elements just as well.)

By the way: The m, n, k and l in the examples may be any amount of FORTH that puts a number on the stack.

Internally making a picture of a compiled CASE structure: (e.g.)

```
address contents (at compile time...)
               . .
               . .
               . .
increasing     . .
memory         lit
addresses       m
               lit
                n
              (case)
               xx-$      CASE
               ....
               ....
               ....
               ....
              branch     ESAC
               ee-$
        xx :   lit
                k
              (case)     CASE
               yy-$
               ....
               ....
               ....
               ....
              branch
               ee-$      ESAC
        yy :   lit
                p
              (case)
               zz-$
               ....      CASE
               ....
               ....
               ....
              branch     ESAC                   branch     ESAC
               ee-$               or we          ee-$
        zz :   drop      ENDCASE  find     zz :  drop      OTHER
        ee :   ....               here:          ....
               ....                              ....
               ....                              ....
               ....                              ....
                                           ee :  ....      ENDCAS
                                                 ....
                                                 ....
                                                 ....
```

Instead of the (CASE) cfa's we may find examples of:

(>CASE), (<CASE), (CASES) etc.

there in a more advanced example.

The m, n, k, p here are compiled literals, but there may be all sorts of FORTH compiled there.

Source definitions in fig-FORTH words

```
SCR #100

   CASE control structure                            AK-80Feb29 )
   ZBRAN              (Bi-level branch if BOT is zero, used by CASE *)
        RP@    2+  SWAP
        IF SWAP DROP 2 ELSE DUP @ @ THEN SWAP +! ;
   COBR                              ( Complete a pending forward branch *)
        HERE OVER - SWAP ! ;
    CASE)    (Compiled by CASE, do a test and conditionally branch *)
        OVER = ZBRAN ;
   CASE        (Execute until ESAC if Key-2 equals Case-1 *)
        ?COMP COMPILE (CASE) HERE 0 , 5 ; IMMEDIATE
   ESAC         (Close a CASE; Key is left if case not done*)
        5 ?PAIRS
        COMPILE BRANCH HERE 0 , SWAP COBR 4 ; IMMEDIATE
   OTHER            (After last ESAC, if stack or nested CASEs used there *
      4  ? PAIRS  COMPILE  DROP   6  ; IMMEDIATE
```

```
SCR #101

   CASE control structure                      AK-80Feb 29 )
   ENDCASE              (Close a CASE control structure  *)
       DUP 4 = IF COMPILE DROP ELSE 6 ?PAIRS 4 THEN
       BEGIN  DUP  4  =  SP@  CSP  @  <  AND
       WHILE  DROP  COBR
       REPEAT  ;  IMMEDIATE
  ;S
```

An 'English' explanation of how the words work:

ZBRAN

Finds 'true' or 'false' on the stack. It fetches the address of the second return stack number which is the pointer (stored IP) in the list where a branch can occur.

- If 'true' was on stack, the pointer is incremented by 2 (making next skip to the CFA following the branch) and the second stack number is dropped. (It was the 'key' to the 'case'.)
- If 'false' was on stack, the pointer is incremented by the value that is found in the location where it is pointing to (making NEXT to resume interpretation of the list where the branch was compiled at a new location).

COBR

Finds an address on stack; a distance from the actual DP value to that address is compiled in the address.

(CASE)

Finds two numbers on stack, compares these and leaves the 2nd on stack. Then control is transferred to ZBRAN. The CFA of (CASE) appears in compiled lists as a relative branch (the relative jump following it in the list).

CASE

Has precedence and checks whether we are compiling when we use it. It compiles (CASE) and puts the address following (CASE)'s CFA in the list, on stack. It stores a temporary 0 in that location and puts a 5 for pair checking on the stack.

ESAC

Does a pair check on the 5 it expects from CASE. It compiles a BRANCH, puts a temporary 0 in the location following that BRANCH and puts the address of that location on stack.

The branch that is half made by the previous CASE is completed. For pair check by the following ENDCASE a 4 is put on stack; the change of check digit (from 5 to 4) makes the nesting of other case structures in CASE .... ESAC possible. (ESAC has precedence.)

OTHER

Has precedence, it does pair check on the 4 it expects from ESAC. It compiles a DROP for the key (for the CASES that are all not fulfilled when this point is reached). The check digit 6 is put on stack. The change from 4 to 6 as a check digit signals to ENDCASE that the 'OTHER' is used and it makes nesting of other case structures in OTHER .... ENDCASE possible.

ENDCASE

Checks for a 4 on stack; in case there is a 4 the "OTHER" is not used and we must compile a DROP here. If there is not a 4 there must be a 6 (which is checked); it is replaced by a 4.

The rest of ENDCASE looks for 4's on stack that are placed there by the previous ESAC's (since there may be 4's on stack already before the definition that contains the case structure. ENDCASE also checks SP@ against CSP contents).

The incomplete branches from the ESAC's are completed until none is left.

Glossary entries for each word in sheet B

```
ZBRAN mf ---     (if f is true)
      mf --- m  (if f is false)
```

Procedure to perform the branch for a high level run time conditional branch in a CASE control structure.

If f is false (zero), the in-line parameter following the compiled reference to the run time conditional branch is added to the stored interpretive pointer (second word on the return stack) to effectuate a branch.

If f is true, 2 is added to skip the in-line parameter and m is dropped. Used by (CASE), (>CASE), (CASES) etc.

COBR addr1 ---

Calculate the branch offset from addr1 to HERE and store in addr1, thus resolving a pending forward branch.

```
(CASE) m n ---     (if m equal n)       C2
       m n --- m (if m unequal n)
```

The run time procedure to conditionally branch in a CASE control structure.

If m equals n, no branching occurs and NEXT interprets the words following the branch offset in the dictionary after the CFA of (CASE).

If m is unequal to n, m remains on stack and NEXT resumes interpretation with a new interpretive pointer value according to the branch offset.

Compiled by CASE. For branching, 2BRANCH is used.

```
CASE m n ---    (run-time, if
                m equal to n)        P,C2
     m n --- m (run-time, if
                m unequal to n)
         --- addr c (compile)
```

Occurs in a colon-definition in the form:

```
   CASE ........... ESAC
   CASE ........... ESAC
     ....           ....
   CASE ........... ESAC
   ENDCASE
or:
   CASE ........... ESAC
   CASE ........... ESAC
     ....           ....
   CASE ........... ESAC
   OTHER .......... ENDCASE
```

At run-time CASE selects execution based on an equality test of the two numbers on stack. If m equals n the part until the next ESAC is executed and then control is passed to after ENDCASE. If m is not equal to n, m remains on the stack and control passes to after the following ESAC. The use of OTHER and its 'other' part are optional. ENDCASE, or (if present) OTHER, drops the remaining m.

At compile time CASE compiles (CASE) and reserves space for an offset at addr. addr and c are used later for resolution of the offset and error testing.

```
ESAC addr1 c1 --- addr2 c2
     (compiling)                    P,C2
```

Occurs within a colon-definition in the form:

```
   CASE .......... ESAC
   CASE .......... ESAC
     ...           ...
   CASE .......... ESAC
   ENDCASE
or:
   CASE .......... ESAC
   CASE .......... ESAC
     ...           ...
   CASE .......... ESAC
   OTHER ......... ENDCASE
```

At run-time ESAC executes after the part selected by the CASE it pairs to. ESAC branches over the following cases and resumes execution after ENDCASE.

At compile time ESAC compiles a BRANCH, reserving room for a branch offset at addr2, leaving addr2 and c2 for later resolving of the offset and error checking. ESAC also resolves the pending branch from the previous CASE at addr1, storing the ofset from addr1 to HERE.

```
OTHER m  ---      (run-time)          C,P
      c1 --- c2   (compiling)
```

Occurs within a colon definition in the form:

```
   CASE ........... ESAC
   CASE ........... ESAC
     ...            ...
   CASE ........... ESAC
   OTHER .......... ENDCASE
```

At run-time OTHER executes when none of the cases is met. OTHER drops the m against which the cases were tested.

At compile time OTHER compiles a DROP. OTHER also checks the c1 from the last ESAC for error testing and puts c2 on stack to signal ENDCASE that OTHER has been used and to make nesting of new case structures possible between OTHER and ENDCASE.

```
ENDCASE addr1 c1 addr2 c1 .....
        addrn-1 c1 addr-n c2 ---
        (compiling)                P,C
        m --- (run-time, no OTHER used)
```

Occurs in a colon-definition in the form:

```
   CASE ........... ESAC
   CASE ........... ESAC
     ...             ...
   CASE             ESAC
   ENDCASE
or:
   CASE ........... ESAC
   CASE ........... ESAC
     ...             ...
   CASE             ESAC
   OTHER .......... ENDCASE
```

At run-time ENDCASE serves as the destination of all forward branches from the ESAC's in the case structure. If OTHER does not occur, ENDCASE drops the m that remained on stack when no case is met.

At compile time ENDCASE compiles a DROP if OTHER was not used in the case structure, which can be known from the value of C2. ENDCASE resolves all the pending forward branches from the ESAC's by storing the offset from addri to HERE in addri for addr1 thru addrn. The C1's indicate the presence of such an unresolved branch as long as the control stack pointer is not passed.

Examples of the use of this statement

```
SCR #102

( Examples of CASE control structure                  AK-80Feb 29 )
: SEL               ( Write number -1 between 0 and 9 as text  " )
     0 CASE  ."  Zero"    ESAC
     1 CASE  ."  One"     ESAC
     2 CASE  ."  Two"     ESAC
     3 CASE  ."  Three"   ESAC
     4 CASE  ."  Four"    ESAC
     5 CASE  ."  Five"    ESAC
     6 CASE  ."  Six"     ESAC
     7 CASE  ."  Seven"   ESAC
     8 CASE  ."  Eight"   ESAC
     9 CASE  ."  NINE"    ESAC
     ." Outside range 0-9" ENDCASE ;


--
```

```
SCR #103

( Examples of CASE control structure                          AK-80Feb29 )
: EXAMPL ( Some text is printed, selected by number -1 & number -2 ")
  5 CASE 12 CASE   ."  This"         ESAC
            3 CASE   ."  is"           ESAC
              OTHER  ."  only"         ENDCASE ESAC
  18 CASE 2 CASE   ."  a very"       ESAC
            7 CASE   ."  silly"        ESAC
              OTHER  ."  example"      ENDCASE ESAC
    OTHER 2 CASE   ."  of the use"   ESAC
            9 CASE   ."  of nested"    ESAC
              OTHER  ."  cases"        ENDCASE ENDCASE ;

; S



```

Short discussion on the case statement presented here

The history of this case statement is outlined below:

A first try for CASE was a replacement of IF so we could produce case structures like:

```
Ia.  ... CASE .... THEN ...
     ... CASE .... THEN ...
     ... CASE .... THEN DROP ...
or
ib.  ... CASE .... ELSE
     ... CASE .... ELSE
     ... CASE .... ELSE DROP
         THEN THEN THEN
```

At run-time, Ib is the faster one since no other cases are tested once a case is done. A disadvantage however is the necessity to write the THEN ... THEN ... THEN series.

An improvement on Ib was to organize the DROP THEN THEN ... THEN by a new compiler word:

```
II.  ... CASE .... ELSE
     ... CASE .... ELSE
     ... CASE .... ELSE
         ENDCASE
```

But since ENDCASE can only see by the 2's for ?PAIR on stack how many branches have to be completed this structure II cannot be nested inside ... IF ... ELSE ... THEN or inside an other CASE ... ELSE.

This could be avoided by making a new "ELSE" and then using other numbers for ?PAIR checking in the structure.

By changing the number for pair checking in the new "ELSE" (ESAC), also nesting in other case structures is possible:

```
III.   ... CASE   ? PAIRS "a"   ESAC   "?PAIRS "b"
       ... CASE                 ESAC
       ... CASE                 ESAC

    ?PAIRS "b"   ENDCASE
```

Now, a remaining problem is the part between ESAC (the last) and ENDCASE. There the number against which the cases are checked is still on stack, so we cannot easily manipulate the stack there; also, at compile time we have the "b"'s for ?PAIR checking on stack so we cannot nest a new case structure there.

To solve these two problems we made the optional "OTHER" that performs the DROP at run time and that at compile time changes again the "check number" to inform ENDCASE that the DROP already has been compiled and to make nesting of other case structures possible.

Of course the nesting problem could have been solved by using an opening word like is done in the example on page 50 and 51 of Forth Dimensions, Vol. 1, No. 5. But this forces the use of an extra word at compile time. This opening word could e.g. store the top stack word on the return stack (not in a variable as is done in the example!, since this prohibits nesting of case structures). But I doubt whether it is an advantage to remove the number against which the cases are checked from the stack: This costs (run) time, makes it difficult to change that number between an ESAC and the next CASE (after all, why should one not be allowed to do this) and the number is not in the way as far as I can see.

The use of a high level (CASE) and the use of the separate ZBRAN there are not mandatory.

To have a fast executing case structure one may rewrite (CASE) in low level without affecting the essence of this case structure.

However, as presented here the structure is machine independent for standard fig-FORTH's.

Also this high level (CASE) makes it easy to extend the possibilities, e.g.:

```
: (>CASE) OVER < ZBRAN ; (BRANCH IF SMALLER THAN)
: >CASE COMPILE (>CASE) HERE 0 , 5 ; IMMEDIATE
```

and we have a new type of case.

or:

```
: (CASES) ROT >R R < SWAP R > AND R>
          SWAP ZBRAN ; (BRANCH IF NOT IN RANGE)
: CASES COMPILE (CASES) HERE 0 , 5 ; IMMEDIATE
```

etc. Any odd case you expect to use more than once can be incorporated in the set and used just like CASE.

;S

P.S. In re-reading all this I notice that "?COMP" is not needed in the definition of CASE; please omit.

P.P.S. My native language is Dutch; please forgive me any errors in the language.

Arie Kattenberg
Utrecht, Netherlands

Judges' Comments - This entry has a number of interesting ideas in it and could be useful to developers. The presentation is a bit hard to follow in places. A plus is the short history of the development of this CASE structure through several earlier forms.

=CASE CONTEST STATEMENT=

George Lyons

This entry submitted to the FIG Case Statement Contest is limited to providing a compiler syntax for writing equivalents of ALGOL "case" and "switch" statements in FORTH and some additional words to use in conjunction with ALGOL style case expressions. As such it does not solve all the problems posed in the contest announcement.

In formulating a case expression syntax the first decision was to treat case lists as in-line or literal expressions within : definitions rather than provide a special defining word creating words of a case list type. This increases flexibility of use at the expense of storage saving otherwise obtainable by exploiting the code address field of a case-type word. A second decision was to allow use of a list either to execute a case selected at run time or to compile the execution address of the case--for use in more complex compiler features. Storage for a list which was to be only executed turned out less than when compiling so different commands are provided for these two circumstances. A third decision was to include in the compiled code for a case list no number-of-cases parameter; hence no checking of the run time inputsubscript's validity is done in executing the cases. Instead separate words ?INDEX and EXCEPT are provided to do this checking, taking more storage when used than if their functions were built into the case list code, but saving time and space when they are not needed, as when the validity of the input is established elsewhere in a program.

The in-line case lists are handled as one instance of a general approach to in-line list functions in which a list is represented in the form ccc(...list data...). cc(is a Word which begins compiling the list and) is introduced as a word, in addition to its role in comments, to terminate the list. Different words ccc(perform different functions involving data or code stored in the list. The parenthesis was defined as a word because of the similarity of the run time process of skipping around data embedded within a definition and the compile time skipping past a comment in source code. The general approach compiles all lists with an execution address at the front which processes the data and returns controls to the point following the list; the address of this return point is stored immediately following the execution address at the beginning, and it has more uses than just returning control. When a list contains variable length elements a vector of addresses of the elements is appended to the end of the list in reverse order. The case lists are an example of this structure in which the data is a list of variable length code segments, written for instance using EXECUTE(CASE case 0 code... CASE casel code... CASE ...). The case compiling words such as EXECUTE(are written using utility Words available for building additional functions along the same lines.

Examples of the latter are some Words that might be used in conjunction with EXECUTE(...). These are mentioned briefly here, and some are implemented in the glossary and code section.

[n] EXCEPT EXECUTE(...)
(compile time source)

At run time EXCEPT will check the subscript on the stack intended to select a case in EXECUTE(...) and replace it by zero if negative or greater than n. Case zero can then be especially written to handle these exceptions.

W+ addr n --- addr+2*n

Address arithmetic operation for byte addressable computers. Increments an address adr by n words. Can be implemented in machine code using shift operations instead of multiply.

```
: W+ 2 * + ;
```

W- addr n --- addr-2*n

Address arithmetic operation for byte addressable computers. Decrements an address adr by n words. See W+.

COMPILES --- addr n P,C

Used in conjunction with EXECUTES in a : definition to combine a procedure performing compiler operations with run time code for the procedure compiled, in a single definition. In the form : ccc ... COMPILES ... EXECUTES ... ; IMMEDIATE, ccc performs the operations up to EXECUTES at compile time; these compile time operations include COMPILES which compiles the address of the code following EXECUTES. EXECUTES places at that address a pointer to the code for : definitions, so that the code following EXECUTES is in effect a : definition without a name field.

```
:COMPILES  ?COMP  COMPILE  COMPILE HERE n 2 ALLOT ; IMMEDIATE
```

EXECUTES addr n --- P,C

See COMPILES. Compiles ;S to terminate the compile time part of a dual definition, and stores the address of the next dictionary location in the space reserved by COMPILES. Compiles the address of the code for : definitions to begin the run time part of the Word.

```
: EXECUTES ?COMP [ HERE 2 - ] COMPILE ;S  n  ?PAIRS HERE
  SWAP ! COMPILE [ @ , ] ; IMMEDIATE
```

```
[ n ] ?INDEX EXECUTE( ... )
(compile time source)
```

At run time ?INDEX will issue a system error message if the subscript on the stack intended to control EXECUTE(is invalid, instead of writing a special case zero in the case list.

```
FIND( n1 , n2 , n3 , ...)  EXECUTE( ... )
```

FIND(is another example of the in-line expression approach which performs the inverse of a simple vector. It searches at run time for a match to the stack item in the list, returning either its subscript or zero if not found. Again, case zero can be written to handle the exceptions.

```
INTERVAL( n1 , n2 , n3 , ... nk )
EXECUTE( ... )
```

Another in-line expression type, INTERVAL(contains a vector of values in ascending order dividing the number domain into the intervals between them. At run time a subscript is returned identifying the interval in which the stack item falls, and the item itself is preserved for processing by the selected case.

```
RANGE( n1 , m1 , n2 , m2 , ... )
EXECUTE( ... )
```

Similar to INTERVAL(except that each n,m pair defines a separate range, and a subscript is generated identifying the first range found which embraces the stack item, or a zero if outside all of the ranges.

n MENU ccc EXECUTE(...) ;

MENU is a defining Word to create a menu-driven application named ccc which at run time will present screen n to the user, who will select options by entering a number, which is finally processed by the case list compiled within ccc.

Glossary and Code

The implementation below is written entirely in high level code assuming a byte addressing machine. Literal "n" used with ?PAIRS is left unspecified for consistent specification of all ?PAIRS values.

BEGIN(--- addr n ff n

Used in certain compiling words to begin compilation of an in-line, or literal data structure within a : definition. The next word in the dictionary is reserved for the address of the location following the entire structure, to be filled in by) at address addr. n is for compiler error checking. ff marks the stack so that other compiling words may push pointers to internal parts of the data block, to be appended to the end of the block by). See).

```
: BEGIN ?COMP HERE n 0 2 ALLOT . ;
```

) addr n ff addr0... n ---
(when used as a Word)

Has two entirely different uses. One terminates a comment begun by (, in which case it is not processed by the compiler. When used outside of a comment it completes compilation of an in-line data structure begun by BEGIN(. addr0... is a possibly empty list of addresses of points internal to the data block left by other compiling Words; if present it is appended to the data in reverse order. The address of the location following the data is then stored back at its beginning point addr. Also resumes compilation mode.

```
:  ) n ?PARIS BEGIN  - DUP WHILE  , REPEAT n ?PARIS HERE SWAP
   ! ] ; IMMEDIATE
```

LIT(---

Used in Words processing in-line data structures to set up the return and computation stacks for accessing the data and branching around it. The Word in whose definition LIT(appears must be used immediately in front of an in-line data block, so that the address of the location at which to resume control is found in the following location; see BEGIN(. Consequently on entry to LIT(the return stack contains the address of the code following LIT(itself on top and the address of the data block just below. LIT(replaces the second return stack item by the address of the code following the data, and pushes the address of the first data item onto the computation stack. Also see)LIT.

```
: LIT( R> R> DUP @ >R 2+ SWAP >R ;
```

)LIT ---

Similar to LIT(except returns on the computation stack the address of the last word in the data structure instead of the first word, for accessing any address vector stored there in reverse order by).

```
: )LIT R> R> @ DUP >R 2 - SWAP >R ;
```

EXECUTE(--- addr n ff n (compile) P,C
n --- (run time)

Used within a : definition to define a list of routines, or cases in high level code in the form:

```
EXECUTE( CASE case0... ;S CASE case1... ;S ... ;S )
```

At run time case n is executed and control returns beyond the list. Unpredictable results occur if n is not a valid subscript at run time. Executes BEGIN(at compile time.

```
EXECUTE( ?COMP COMPILES BEGIN( EXECUTES )LIT W- @ >R ; IMMEDIATE
```

CALL(--- addr n ff n   (compile)   P,C
        n ---   (run time)

Similar to EXECUTE(except the case routines are in assembly language in the form CALL(CASE case0... CASE case1... ...). Invokes the assembler vocabulary and suspends compilation.

```
CALL( ?COMP COMPILES BEGIN( [COMPILE] [ [COMPILE] ASSEMBLER
EXECUTES )LIT W- @ SP@ 2 - SWAP DROP EXECUTE; IMMEDIATE
```

CASE   addr n ff addr0... n ---
       addr n ff addr0...addr1 n   P

Used to begin each cae in a case list defined by EXECUTE(or CALL(. Adds the address of the next case addr1 to the list of case addresses addr0... on the stack, using n for error checking.

```
: CASE n ?PAIRS HERE n ; IMMEDIATE
```

COMPILE(--- addr n ff n (compile)   P,C
           n --- addr   (run time)

Used within a : definition to define a list of routines, or cases in either machine code or high level code in the form COMPILE(:CASE ... ;S ... CODECASE ...) which returns at run time the execution address of the case whose subscript is on the stack. The input subscript must be valid or unpredictable results will occur. To actually compile the execution address returned use ,. See also EXECUTE(and LITERAL(. Compiles using BEGIN(.

```
COMPILE( ?COMP COMPILES BEGIN( EXECUTES )LIT W- @ ; IMMEDIATE
```

:CASE   addr n ff addr0... n ---
        addr n ff addr0... addr1 n   P

Used to begin each high level code case in a case list defined by COMPILE(. Executes CASE and compiles the address of the code for executing : definitions. The routine begun by :CASE should be terminated by ;S as in EXECUTE(expressions.

```
: :CASE [ HERE 2 - ] [COMPILE] CASE ] COMPILE [ @ , ] ; IMMEDIATE
```

CODECASE   addr n ff addr0... n ---
           addr n ff addr0...addr1 n   P

Used to begin each assembly code routine in a case list defined by COMPILE(. Executes CASE and compiles the address of the next dictionary location (as in the code field for a CODE definition). Compilation is suspended and the assembler vocabulary invoked as in CALL(. A jump to NEXT within a machine code case will resume high level execution following the case list.

```
: CODECASE [COMPILE] CASE 2 ALLOT HERE DUP 2 - !
  [COMPILE] [ [COMPILE] ASSEMBLER ; IMMEDIATE
```

LITERAL(--- addr n ff n (compile)
           n1 --- n2   (run time)   P,C

Used within a : definition to define a vector of 16-bit values. These values may be made Word execution addresses using the form

```
LITERAL( Word0 Word1 Word2 ... )
```

or may be made literal numbers using the form

```
LITERAL( [ n0 , n1 , n2 , ...] )
```

At run time the element whose subscription is on the stack is returned (without checking the validity of the stack value). When used with EXECUTE in the form LITERAL(...) EXECUTE the same result is achieved as using EXECUTE(...) except that storage requirements are less because no extra addresses are needed at the end of the vector. Uses BEGIN(to compile the list.

```
:LITERAL( ?COMP COMPILES BEGIN( EXECUTES LIT( W+ @ ; IMMEDIATE
```

EXCEPT n --- (compile)
n --- n (run time) P,C

Used before an in-line case list or literal vector defined by EXECUTE(...) or similar Words, in the form [n] EXCEPT. Compiles an execution address and the value n, presumed to be the number of caes or vector elements in the subsequent in-line expression. At run time replaces any input value that is negative or greater than n by zero, allowing case or element zero to represent the "exceptions." This may be an error message or other explicit operation, or may simply bypass the entire case list by leaving case zero empty, i.e. compiling high level cases as ;S and machine code cases as a jump to NEXT. The EXCEPT function is not built into the case list expression code itself to allowing saving the storage when it is not needed.

```
: EXCEPT >R COMPILES R> , EXECUTES R> DUP 2+ >R OVER 0< IF
  DROP DROP 0 ELSE @ OVER < IF DROP 0 ENDIF ENDIF ; IMMEDIATE
```

FIND(--- addr n ff n (compile)
nl --- n2 (run time) P,C

Used in a : definition to define an array similar to LITERAL(but to perform the reverse operation at run time, i.e. the value is on the stack and the subscript is returned, or zero if not found.

```
: FIND( COMPILES BEGIN( 2 ALLOT ( element zero reserved )
( for copy of input) [COMPILE] [ EXECUTES LIT(
OVER OVER ! SWAP OVER R 2 - DO DUP I @ = IF DROP I LEAVE
ENDIF -2 +LOOP SWAP - 2 / ; IMMEDIATE
```

INTERVAL(--- addr n ff n (compile) P,C
nl --- nl n2 (run time)

Used in a : definition to define a literal vector of interval boundary points in increasing order; at run time the subscript of the smallest boundary above nl is added to nl already on the stack, to control a subsequent case list processing nl. Compiles using BEGIN).

```
: INTERVAL COMPILES BEGIN( [COMPILE] [ EXECUTES ] LIT(
OVER OVER R 2 - DO I OVER I @ < IF LEAVE ENDIF
2 +LOOP SWAP DROP SWAP - 2 / ; IMMEDIATE
```

APPENDIX

The words COMPILES and :CASE above share a common function which might preferably be in a separate Word by itself. That function is compiling into the next dictionary location the code address used in : definitions. Rather than define a new Word, however, this function may be added to the existing definition of the : operator, as the function to be performed when STATE is the compiling mode, in contrast to the regular function performed when STATE is the execution mode, as it is when a definition is begun using :. Similarly, the Word CODE can be expanded to include a function to be performed in the compile state which consists of compiling the code address of a CODE definition (the address of the following location...), setting STATE to execute and invoking the ASSEMBLER vocabulary for beginning assembly language programming immediately following. Revised definitions from the case statement glossary above would then be:

```
: EXECUTES ? COMP COMPILE ;S n ?PAIRS HERE SWAP !
  [COMPILE] : ; IMMEDIATE
```

The Words :CASE and CODECASE are eliminated and the syntax for COMPILE(is:

```
COMPILE( CASE : ...high level case...;S ... )
COMPILE( CASE CODE ...machine code case...  .. )
```

George Lyons
Jersey City, NJ 07302

Judges' Comments - George got off to a great start but went on to solve many more problems than CASE, i.e. compiling in-line machine code by CODECASE. There are numerous ideas here, deserv- ing of further analysis and examples o CASE.

=A FORTH CASE STATEMENT=

R. D. Perry

The Case Statements presented here are an extension of the FORTH IF Statement. The structure of the CASE Statement is such that it allows an N-way branch as contrasted to the IF statement two way branch. This version allows a CASE to be tested against a single value or a range of values. It does not require contiguous values for the tests. The value or range of values to be tested against are determined at run-time, this allows variables to determine CASE selection. No preprocessing is required as with the vector selection approach. It will execute faster than an IF statement preceeded by preprocessing (Example: = IF) assuming code implementation of N=Branch and NRANGE=BRANCH.

I became interested in the CASE Statement while implementing a CRT Screen Editor for FORTH Editing and Word Processor use.

```
SCR # 81
( CASE STATEMENTS        RDP 800322  )
HEX
CODE N=BRANCH ( IF BOT NOT EQU SEC BRANCH FROM INLINE LITERAL )
   INX, INX, FE ,X LDA, BOT CMP, 0=
    IF, FF ,X LDA, BOT 1+ CMP, 0=
       IF, INX, INX,  ' 0BRANCH 8 +  ( BUMP ) JMP,
       ENDIF,
    ENDIF,  ' BRANCH JMP, C;

CODE NRANGE=BRANCH ( IF THIRD<SEC OF THIRD>BOT BRANCH FROM LIT )
   INX, INX, INX, INX, SEC, FC ,X LDA, BOT SBC,
   FD ,X LDA, BOT 1+ SBC, 0< NOT
      IF, SEC, BOT LDA, FE ,X SBC, BOT 1+ LDA, FF ,X SBC, 0< NOT
         IF, INX, INX,  ' OBRANCH 8 +  ( BUMP ) JMP, ENDIF,
      ENDIF,  ' BRANCH JMP,  C:
DECIMAL -->
```

```
SCR # 82

( CASE STATEMENTS        RDP 800322
 -->    ( REMOVE THIS LINE IF CODE VERSIONS NOT USED )
DECIMAL    ( R IS POINTING TO NEXT LOCATION )
: N=BRANCH
   OVER =
   IF R> 2+ >R DROP
   ELSE R> DUP   @ + >R
   THEN  ;

: NRANGE=BRANCH
   ROT DUP ROT ( L,V,V,H ) >
   IF SWAP DROP R> DUP @ + >R ( OVER RANGE )
   ELSE DUP ROT ( V,V,L ) <
      IF R> DUP @ + >R ( UNDER RANGE )
      ELSE R> 2+ >R DROP ( IN RANGE )
  THEN THEN ; -->
```

```
SCR # 83

( MORE CASE        RDP 800322  )
: BEGIN-CASES   ?COMP   0  4 ; IMMEDIATE

: CASE    ?COMP ( EL,4 ) 4 ?PAIRS ( EL )
   COMPILE N=BRANCH  HERE  0 ,  ( EL,NBL  )
   5 ; IMMEDIATE  ( EL, NBL,5 )

: RANGE-CASE  ?COMP ( EL,4 )   4 ?PAIRS    ( EL )
   COMPILE NRANGE=BRANCH  HERE   0 ,       ( EL,NBL )
   5 ; IMMEDIATE  ( EL,NBL,5)

: ELSE-CASE  ?COMP 4 ?PAIRS ( EL )
   COMPILE DROP  0  5 ; IMMEDIATE   ( EL,0,5 )
-->
```

```
SCR # 84

( MORE CASE        RDP 800322  )
: END-CASE   ?COMP  5 ?PAIRS  COMPILE BRANCH ( EL,BL )
    DUP ( EL,BL,BL, )
   IF HERE  2+  OVER - SWAP ! ( EL )
   ELSE DROP
   THEN  HERE  SWAP  ,   4 ( NEL,4 ) ;   IMMEDIATE

: END-CASES ?COMP 4 ?PAIRS  ( EL )
   DUP 0= 0= 1 ?PAIRS ( ERROR IF NO CASES )
   COMPILE DROP
   BEGIN  DUP
   WHILE  DUP @ SWAP HERE OVER -  SWAP !
   REPEAT DROP ; IMMEDIATE      ;S
( NAMES OF STACK ITEMS )
EL -> END LINK              NEL -> NEW END LINK
BL -> BEGIN LINK            NBL -> NEW BEGIN LINK
```

```
SCR # 85
( CASE STATEMENT TEST          RDP 800320 )
: TEST BEGIN-CASES
      1  CASE  " ONE" END-CASE
      2  CASE  " TWO" END-CASE
      -9 9  RANGE-CASE  " > NEG.  TEN AND < TEN" END-CASE
      ELSE-CASE  " OTHER" END-CASE
    END-CASES  CR ;
;S

TYPE A NUMBER FOLLOWED BY "TEST", OUTPUT WILL BE
ACCORDING TO CASE ABOVE
```

```
N=BRANCH n1 n2 --- (run-time, n1=n2)
         n1 n2 --- (run-time, n1<>n2)
```

The Run-Time procedure to conditionally branch. If n1 does not equal n2 the following In-Line parameter is added to the interpretive pointer to branch ahead (or back) and n2 is dropped. If n1 equals n2 the interpretive pointer is advanced passed the in-line parameter and both n1 and n2 are dropped. Compiled by CASE.

```
NRANGE=BRANCH n1 n2 n3 ---
              (run-time, n1>=n2 & n1<=n3)
              n1 n2 n3 --- n1
              (run-time, n1<n2 or n1>n3)
```

The Run-Time procedure to conditionally branch. If n1 is less than n2 or n1 is greater than n3 the following in-line parameter is added to the interpretive pointer to branch ahead (or back) and both n2 and n3 are dropped. If n1 is greater than or equal to n2 and n1 is less than or equal to n3 and n1, n2, and n3 are dropped and the interpretive pointer is advanced passed the In-Line parameter. Compiled by RANGE-CASE.

```
BEGIN-CASES --- n1 n2
           (compile time)
```

Occurs in a colon-definition in the form:

```
BEGIN-CASES
... CASE ... END-CASE
... RANGE-CASE ... END-CASE
ELSE-CASE --- END-CASE
END-CASES
```

At compile-time BEGIN-CASES places n1 and n2 on the stack. n1 will later be used by END-CASES to signal that there is no prior END-CASE to link to. n2 is used for error testing.

```
CASE n1 n2 --- n1 (routine, N1<>n2)
     n1 n2 ---    (routine, n1=n1)
addr1  N1 --- Addr1 Addr2 N2
              (compile time)
```

At Run-Time CASE selects execution based on equality of the bottom two values on the stack. If they are equal signalling that the CASE is to be executed, both n1 and n2 are dropped and execution proceeds through CASE. CASE. If they are not equal only N2 is dropped and execution skips to just after END-CASE. (See BEGIN-CASES)

At Compile-Time CASE compiles N=BRANCH and reserves space for an offset value at addr2. addr1 is the address for the offset value of the last END-CASE. n1 and n2 are used for error testing.

```
RANGE-CASE
   N1 N2 --- N1 (Run-Time, N1<>N2 ) P,C2
   N1 N2 ---    (Run-Time, N1=N2)
addr1 N2 --- addr1 addr2 N2 (Compile-Time)
```

At Run-Time selects execution bases on whether n1 is in the range n2 to n3 (n2<n3). If in range execution proceeds through RANGE-CASE. If not in RANGE execution skips to just after END-CASE. (See BEGIN-CASES)

At Compile-Time RANGE-CASE compiles NRANGE=BRANCH and reserves space for an offset at addr2. addr1 is the location for the offset value of the last END-CASE. n1 and n2 are used for error testing.

```
ELSE-CASE
      n1 --- (run-time)
addr1 n1 --- addr n2 n3
             (compile-time)
```

At Run-Time n1 is dropped and execution continues through ELSE-CASE. (See BEGIN-CASES)

At Compile-Time ELSE compiles DROP. ADDR is the location for the offset of the last END-CASE. n2 is used by END-CASES to signal that the last case was an ELSE-CASE. n1 and n3 are used for error testing.

```
ELSE-CASE --- (run-time)
  addr1 addr2 n1 ---
        addr3 n2 (compile time)
```

At Run-Time causes execution to skip to after END-CASES. (See BEGIN-CASES)

At Compile-Time uses ADDR2 to set the offset of the last CASE or RANGE-CASE to point to after this END-CASE. Compiles BRANCH with an offset to be calculated later by END-CASES. The location for the offset of the last END-CASE is temporarily stored in this offset location and the new offset location is put on the stack. N1 and N2 are used for error testing.

```
END-CASES
           --- (run-time with ELSE-CASE)
         n --- (other run-time)
addr1 n1 --- (compile-time)
```

At Run-Time drops a stack value if no ELSE-CASE exists. Any END-CASE will continue execution just after the DROP. (See BEGIN-CASES)

At Compile-Time DROP is compiled and all of the offsets from an END-CASE are calculated and stored in their proper locations. ADDR is the location for the last offset for an END-CASE. That location holds the address for the prior offset and so on. The first offset location holds a value (0) which tells END-CASE that there are no more offsets to calculate.

R.D. Perry
San Diego, CA 92106

Judges' Comments - This is quite a complete and well documented entry. The range-of-cases feature is well done. Note that high level alternatives are given for the 6502 machine CODE words.

NEW PRODUCT

pico FORTH

HERMOSA BEACH, CA, JUNE 24, 1980 picoFORTH^{TM}, a new subset of polyFORTH^{TM}, is available for 1802 (disk or PROM) and 8080 micro- processors. Designed for interactive evaluation, picoFORTH includes all the essentials for programming, debugging, and testing a single-task application. This complete operating system features the polyFORTH assembler, compiler, text interpreter, editor, disk utilities, and basic documentation. picoFORTH can be upgraded at any time, either for a single purpose (with one or more of three packages: Source, Target Compiler^{TM}, or Mulititasker) or to full polyFORTH. A File Management Option package is also available. In addition to the current versions, picoFORTH will soon be implemented on the 8086, 6800, and LSI-11 processors. Price for picoFORTH is $495. Write or call Tom at FORTH, Inc., 2309 Pacific Coast Highway, Hermosa Beach, CA 90254 (213) 372-8493.

NEW PRODUCT

ALPHA MICRO FORTH

This system implements the Forth Interest Group language model, with full-length names to 31 characters, and extensive compile-time checks.

In addition, the diskette includes an editor, a FORTH assembler, and a string package, in FORTH source. The PDP-11 FORTH User's Guide, which includes extensive annotated examples of FORTH programming.

This FORTH system runs under AMOS. The distribution disk is single density. The complete system price is $190: Professional Management Services, 724 Arastradero Road, Suite 109, Palo Alto, California 94306, (408) 252-2218.

CASE STATEMENT

William H. Powell

The case structure by R.B. Main looks very powerful and flexible, but it seems to me to be unnecessarily complicated. My suggestion is for a word that does OVER = IF for his word CASE. This fits the existing FORTH compiler very well. The example by Main would read

```
: MONITOR
    41 CASE ." ASSIGN " THEN
    44 CASE ." DISPLAY " THEN
    46 CASE ." FILL " THEN
    47 CASE ." GO " THEN
    53 CASE ." SUBSTITUTE " THEN
       ELSE ." INSERT " THEN
       DROP ;
```

You will note that I have made the 'insert' message unconditional. This illustrates just how little need be added to the present FORTH structure and also how use of the present FORTH conditionals can be harnessed to the simple case structure as above. The normal FORTH syntax holds, and can be relied upon if case structures are nested into other structures, or into another set of case conditions.

This structure is neither the optimum for speed nor bytes. On the other hand we should avoid adding to FORTH in such a way that the nucleus and compiler grow any more than necessary. I favor a CASE structure that makes the program clearer, encourages sound software design and adds power to the language without adding significantly to the system software overhead.

Using the fig-FORTH model I need ideally one more nucleus word, and one for the compiler....

```
CODE /=BRANCH         ( Branch if SEC - BOT non-zero)
      INX, INX,       ( Drop BOT  only)
      SEC, FE ,X LDA,  0 ,X SBC, ' BRANCH 0= END,
           FF ,X LDA,  1 ,X SBC, ' BRANCH 0= END,
           BUMP: JMP,

: CASE ( n1 n2 --- n1 Case is executed if n1 = n2)
    COMPILE /=BRANCH HERE 0 , 2 ; IMMEDIATE
```

You will see that /=BRANCH does the same as OVER = IF and the case structure could be implemented without introducing /=BRANCH but I think speed and clarity better if one adds a cod word as I have.

W.H. Powell
Sawbridgeworth
Herts. CM21 9NB
ENGLAND

Judges' Comments - Bill Powell didn't submit this as a contest entry, but it appeared in our mail just as the contest started. We took the liberty of including it as a mini-Case appropriate for the 6502.

---- HELP WANTED ----

A CASE STATEMENT

Major Robert A Selzer

OVERVIEW OF THE STATEMENT

CASE - The "CASE" statement is a special form of the IF-ELSE-THEN that permits the selection of one of many cases depending upon the top word on the stack being equal to a specified word (the value that precedes "CASE").

u (stack value) ul (case value) CASE (true action) ELSE (false action) THEN

u (stack value) ul (case value) CASE (true action) THEN

If u = ul, drop u,ul and execute true action following CASE until ELSE or THEN. Otherwise, drop ul but leave u on stack and execute ELSE (false action) or THEN if no ELSE. Implementation is the same as IF-ELSE-THEN, however each subsequent use of "CASE" will save 2 words (4 bytes) over the explicit use of OVER = IF DROP. CASE use also improves the readability of the source and if used often, will save code as well as being more convenient to the user.

SOURCE DEFINITIONS

See attached source listing. Note that fig-FORTH word COMPILE should replace FORTH, INC. word (back-slash) or X (in later FORTH, INC versions) and a 16 bit emplace word , (comma) replaces the 8 bit emplace C, (C-comma). So, for SCR # 198, line 6. The fig-FORTH definition for CASE would be:

```
:  CASE  COMPILE (CASE) COMPILE OBRANCH  HERE 00  ,  ; IMMEDIATE
```

ENGLISH EXPLANATION

Only two new words need to be defined to use the CASE statement. (CASE) is the execution version that duplicates (OVER) the top of stack value then compares (=) it to the case value. If they are equal, the true action through the IF statement is taken and the stack value u is dropped (DROP). As part of the true action a flag (1) is pushed on the stack for 0BRANCH to test when CASE is executed. If the stack and case values are not equal the false action (ELSE) is taken and a false flag (0) is pushed on the stack over the original stack value tested (u). Both actions exit with THEN. CASE compiles the address of (CASE) and the address of the run-time IF called 0BRANCH . A 16 bit zero is compiled (,) at HERE in the dictionary, by HERE 00 , to reserve space for the branch to ELSE or THEN. The precedence bit of CASE is set so that CASE compiles 6 bytes whenever it is executed. Like IF, CASE must be used inside a colon definition and each use of CASE requires a corresponding THEN (or ELSE) to complete the structure.

GLOSSARY ENTRIES

(CASE)

The run-time procedure that is used by CASE, Equivalent to OVER = IF DROP. (CASE) is compiled by CASE.

CASE u ul --- u P,C2+

u ul CASE true action for u=ul
ELSE u false action THEN

If u=ul, drop u and ul and execute true action following CASE until next ELSE or THEN. If u is not equal to ul, drop ul but leave u and execute false action following ELSE or drop ul but leave u if no ELSE and exit to THEN.

```
u  ul  CASE  true action for u=ul  ELSE
u  u2  CASE  true action for u=u2  ELSE
u  un  CASE  true action for u=un  ELSE
          u  false action  THEN  THEN...THEN
```

EXAMPLES

See screens #199 and #200.

SCR #199 is used to demonstrate simple CASE use in the same application of the example published in FORTH-DIMENSIONS v 1/5, p. 51 to show conformity to an existing structure.

SCR #200 is a simple, but elegant example of CASE use in a video editor which occupies about 355 bytes of dictionary space for the COMPLETE editor. This is a good example of the CASE structure in fig-FORTH used to save code space and provide clarity of structure. While the editor is written for the ADM-3A terminal, line 1 defines a word which controls the cursor position sequence, so that any terminal can be used by making appropriate changes to the word YXCUR . The integer values in line 2 (2 and 4), determine the initial Y,X offset of the cursor in the HOME position (upper left corner + Y,X offset). This allows for adjustment of different LIST formats and edit screen positions. The vertical line at the right margin of the screens is generated by a 7C EMIT compiled in LIST. This vertical line gives the video editor user a positive indication of the editor limits of the right margin by setting up a window in which to edit. The ESC ($1B) key is used to exit the video editor VEDIT when finished. In fig-FORTH, use EMIT in place of ECHO in line 1. Don't forget to FLUSH .

DISCUSSION

This implementation of CASE in this form is fig-FORTH transportable to different machines (ie., 6502, 8080, 6800 etc.), however there is a 6 byte requirement for each use of CASE versus only 4 bytes for each use of IF. In applications like the example shown in SCR #200, the 2 byte overhead in CASE (6 bytes vs. 4 bytes for IF) saves 4 bytes for each use in lieu of OVER = IF DROP (10 bytes). More importantly, its use significantly enhances the readability and structure of the source code at the minimum cost of only 2 new FORTH words.

```
     SCR# 198

 0 (  CASE DEFINITION        RAS-09FEB80 )
 1 FORTH DEFINITIONS  BASE  @  HEX  FORGET  TASK  :  TASK  ;
 2
 3
 4 :  (CASE )  OVER = IF DROP 1 ELSE 0 THEN  ; ( EXECUTION CODE )
 5
 6 : CASE            (CASE)     0BRANCH  HERE  0  C,  ;  IMMEDIATE
 7
 8 BASE !
 9
10
11
12
13
14
15

     SCR# 199

 0 (  TEST  " CASE "  STRUCTURE )  BASE  @  HEX
 1
 2 :  MONITOR
 3     41 CASE  ." ASSIGN "     ELSE
 4     44 CASE  ." DISPLAY "    ELSE
 5     46 CASE  ." FILL  "      ELSE
 6     47 CASE  ." GO "         ELSE
 7     49 CASE  ." INSERT "     ELSE
 8     53 CASE  ." SUBSTITUTE " ELSE
 9      THEN THEN THEN THEN THEN THEN  ;
10
11 : KEYBOARD  BEGIN  KEY 7F AND DUP MONITOR 20 = END  ;
12
13 BASE !   ;S
14
15

     SCR# 200

 0 ( VIDEO EDITOR, COPYRIGHT RCS 1978    ) HEX   00 VARIABLE  CUR
 1 : YXCUR  1B ECHO  3D ECHO  20 + ECHO 20 + ECHO ; ( ADM-3A)
 2 : .CUR CUR @ 40  /MOD 2 + SWAP 4 + SWAP YXCUR  ; : !CUR  0 MAX
 3   3FF MIN CUR ! ;  : +CUR  CUR @ + !CUR ;  : +.CUR  +CUR .CUR ;
 4 : +LIN CUR @ 40 / ( LINE #) + 40 * !CUR ;  : HOM    00 CUR !  ;
 5 : !BLK SCR @ 8 * CUR @ 80 /MOD ROT + BLOCK + C! UPDATE 1 +.CUR ;
 6   : VEDIT           LIST CR CR CR CR CR HOM .CUR BEGIN
 7           KEY 1B  CASE  0 12 YXCUR QUIT  ELSE  ( ESCAPE)
 8               0B  CASE    -1 +.CUR       ELSE  ( LEFT  CURSOR)
 9               0A  CASE    40 +.CUR       ELSE  ( DOWN  CURSOR)
10               0B  CASE   -40 +.CUR       ELSE  ( UP    CURSOR)
11               0C  CASE     1 +.CUR       ELSE  ( RIGHT CURSOR)
12               0D  CASE     1  +LIN .CUR  ELSE  ( NEW   LINE  )
13               1E  CASE         HOM .CUR  ELSE  ( HOME  CURSOR)
14               DUP ECHO !BLK
15         THEN THEN THEN THEN THEN THEN THEN AGAIN ; DECIMAL ;S

Copyright 1977, RCS Associates                         FORTH-65 Ve
OK
```

Major Robert A. Selzer
APO San Francisco, 96301

Judges' Comments - This entry has the unfortunate need for closing the CASE by a correct number of THENs. It is written for microFORTH. The example of a screen text editor is outstanding and should be carefully read by all.

A CASE STATEMENT

Kenneth A. Wilson

CASE STATEMENT CONTEST

1.0 Description of the entry (coded in microFORTH)

1.1 Screen 338 defines the 4 words needed to generate a complete CASE statement.

1.2 Screen 339 contains a CASE test example.

1.3 The next 2 pages contain the printout obtained by executing the word TRIAL.

2.0 Definition of CASE words

	word	vocabulary	block	stack in	stack out
2.1	<CASE	FORTH	338	1	0
2.2	->	FORTH	338	0	1
2.3	=CASE	FORTH	338	3	0
2.4	CASE	FORTH	338	2	0

2.1 <CASE: A defining word which creates a named array of n + 1 cells. Example: n <CASE name.

2.2 ->: A redefinition of for visual clarity. Pushes onto the stack the address of the parameter field of the word that follows in the current input stream.

2.3 =CASE: Puts the address of a word (S1) into an array (S0) at cell n (S2).

Example: n word array =CASE
Read as: "n" becomes "word" in "array" case.

2.4 CASE: Executes the word whose address is contained in the array (S0) at cell location n (S1).

Example: n name CASE

3.0 Explanation of the Example in Screen 339.

3.1 Line 1 defines 3 Cases:

3.1.1 FIRST is a Case of 4 cells
3.1.2 SEC is a Case of 4 cells
3.1.3 THIRD is a Case of 4 cells

3.2 Lines 2 thru 5 define "printing" words as follows:

3.2.1 Pronouns: I, YOU, WE, THEY
3.2.2 Verbs: RUN, WAL, SIT, JOG
3.2.3 Adverbs: HOME, BACK, DOWN UP

3.3 Line 6 thru 9 define the contents of the three Cases as follows:

3.3.1 FIRST Case contains 4 Pronouns

3.3.2 SEC Case contains 4 Verbs

3.3.3 THIRD Case contains 4 Adverbs

3.4 Lines 10 thru 14 define the word TRIAL which when executed, will cause the three Cases to be executed in sequence for each different possible combination of the index. i.e.:

```
111 FIRST CASE SEC CASE THIRD CASE
112 FIRST CASE SEC CASE THIRD CASE

554 FIRST CASE SEC CASE THIRD CASE
555 FIRST CASE SEC CASE THIRD CASE
```

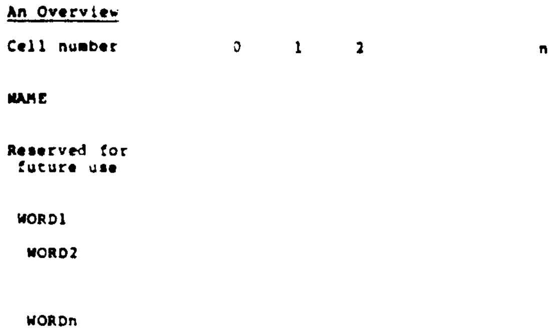


Figure 1

A Case Array NAME of n+1 Cells

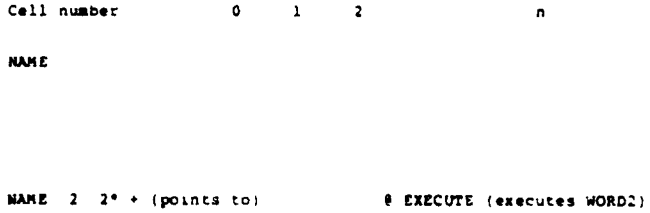


Figure 2

Storing and Executing Cell 2

```
338 LIST
  0  ( CASE TEST WORDS)
  1  DISPLAY DEFINITIONS
  2  : <CASE  0 VARIABLE   2* H +! ;
  3  : ->     ;
  4  : =CASE  ROT 2*  +  !  ;
  5  : CASE  SWAP 2* +  @  EXECUTE ;
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15  DECIMAL      ;S    KAW        2-18-80
OK
```

```
339 LIST
  0  ( CASE TEST EXAMPLE )  DISPLAY DEFINITIONS    DECIMAL
  1  4 <CASE FIRST   4 <CASE SEC     4 <CASE THIRD
  2  : II [ I ] ;  : YOU [ YOU ]  ;  : WE [ WE ] ;  :  THEY [ THEY ] ;
  3  : RUN [ RUN ] ; : WALK [ WALK ]  ; : SIT [ SIT ]  ;
  4  : JOG [ JOG ] ; : HOME [ HOME ]  ; : BACK [ BACK ] ;
  5  : DOWN [ DOWN ] ; : UP [ UP ] ;
  6  1 -> II FIRST =CASE  1 -> RUN SEC =CASE  1 -> HOME THIRD =CASE
  7  2 -> YOU FIRST =CASE 2 -> WALK SEC =CASE  2 -> BACK THIRD =CASE
  8  3 -> WE FIRST = CASE 3 -> SIT SEC =CASE   3 -> DOWN THIRD =CASE
  9  4 -> THEY FIRST =CASE 4 -> JOG SEC =CASE  4 -> UP THIRD =CASE
 10  : TRIAL  CR  5 1 DO I
 11           5 1 DO I
 12           5 1 DO OVER OVER I ROT ROT
 13           FIRST CASE  SEC CASE  THIRD CASE  CR
 14           LOOP DROP CR   LOOP DROP CR   LOOP ;
 15  DECIMAL      ;S      KAW      2-28-80
OK
```

```
TRIAL

I RUN HOME
I RUN BACK
I RUN DOWN
I RUN UP

YOU RUN HOME
YOU RUN BACK
YOU RUN DOWN
YOU RUN UP

WE RUN HOME
WE RUN BACK
WE RUN DOWN
WE RUN UP

THEY RUN HOME
THEY RUN BACK
THEY RUN DOWN
THEY RUN UP

I WALK HOME
I WALK BACK
I WALK DOWN
I WALK UP

YOU WALK HOME
YOU WALK BACK
YOU WALK DOWN
YOU WALK UP

WE WALK HOME
WE WALK BACK
WE WALK DOWN
WE WALK UP

THEY WALK HOME
THEY WALK BACK
THEY WALK DOWN
THEY WALK UP

I SIT HOME
I SIT BACK
I SIT DOWN
I SIT UP

YOU SIT HOME
YOU SIT BACK
YOU SIT DOWN
YOU SIT UP
```

```
WE SIT HOME
WE SIT BACK
WE SIT DOWN
WE SIT UP

THEY SIT HOME
THEY SIT BACK
THEY SIT DOWN
THEY SIT UP

I JOG HOME
I JOG BACK
I JOG DOWN
I JOG UP

YOU JOG HOME
YOU JOG BACK
YOU JOG DOWN
YOU JOG UP

WE JOG HOME
WE JOG BACK
WE JOG DOWN
WE JOG UP

THEY JOG HOME
THEY JOG BACK
THEY JOG DOWN
THEY JOG UP

OK
```

Kenneth Wilson
Waltham, MA 02154

Judges' Comments - This is a very simple positional (jump table) type of CASE. The whole thing can be defined in three short lines of code. At first glance, however, the presentation looks more difficult than it is. Part of the problem is that the notation - the word names - does not suggest, very well, what is going on. This entry looks like a good complement to Eaker's. Both are simple mechanisms for doing a single job and the jobs that they each do are very different. Work is needed on integration and further development of these models.

NEW PRODUCT

68000

CREATIVE SOLUTIONS, INC. announces the availability of the FORTH programming approach for the Motorola 68000 16-bit Microprocessor.

Featuring: FORTH Interest Group Model and FORTH-79 Standard Compatibility, Virtual Disk Operating System, Text Editor, Inline Macro Assembler, Computer Aided Instruction Course on the FORTH Proramming Approach.

Also Available: Customized I/O Drivers for Non-Standard configurations, Suitable Hardware Configurations, Complete Source (written in FORTH), Meta Compiler, Multi-tasker, Extended Data Base Management and File System.

The standard software product, available for configurations utilizing the Motorola MEX68KDM (D2) 68000 evaluation model with Persci 1070 controller and compatible floppy disk drives retails for between $1500 - $5000 (depending upon options) for single user systems.

For further information please contact Creative Solutions, Inc., 14625 Tynewick Terrace, Silver Spring, Maryland 20906, Phone: (301) 598-5805.

NEW PRODUCT

AVAILABLE FROM ANCON

The following manuals and other information is available from ANCON, 17370 Hawkins Lane, Morgan Hill, CA 95037. Write for detailed list.

- FORTH Systems Reference Manual
- The FORTH Language
- FORTH-11 Reference Manual
- Indirect Threaded Code Reprints
- FORTH, a Programmers Guide
- PDP-11 FORTH Users Guide
- PH21-MX FORTH Manual
- CYBOS Programmers Manual
- Program FORTH, A Primer
- The JKL FORTH Manual

CASE STATEMENT

WAYNE WITT/BILL BUSLER

Overview

The CASE word provides the capability to vector to a particular word based on an input parameter, similar to the FORTRAN computed go-to. The CASE word also provides automatic limit checking on the input parameter with an optional out-of-range capability (OTHERCASE).

```
    49
( NEW CASE  -  CODE CASE              WW & WB  2/15/80  ) HEX

CODE I!   Y' PULU   NEXT          ( IP = TOP OF STACK )

: (CASE)                       ( CODE CASE -- CASE PARAMETER N -1 )
   I @ 7FFF AND OVER SWAP <<          ( TRUE IF N IN LIST RANGE )
     IF 2* I + 2+ @ 2+ EXECUTE            ( EXEC. LIST MODULE N )
     ELSE DROP I @ 0<           ( TRUE IF OTHERCASE SPECIFIED )
      IF I @ 7FFF AND 2* I + 2+ @ 2+ EXECUTE       ( OTHERCASE )
     THEN THEN                  ( NOW TO CONTINUE EXEC. AT DONE )
   I DUP @ DUP 0<          ( GET ADDR. AND VALUE OF CASE-INDEX )
     IF 7FFF AND 1+ THEN ( INCR INDEX IF OTHERCASE SPECIFIED )
   2* 2+ + R> DROP I! ;        ( CONTINUE EXECUTION AFTER DONE )

( NOTE: INTERPRETER POINTER MOVED TO END OF LIST OR)
(         AFTER THE DONE )   DECIMAL   ;S
```

```
    50
( NEW   CASE - OTHERCASE - DONE     WW & WB   2/15/80    ) HEX
                         ( PUT CODE CASE ADDRESS IN DICTIONARY , )
                                               ( PUT H ON STACK , )
: CASE    (CASE) HERE 0 , ;  IMMEDIATE    ( CREATE CASE-INDEX )
                                 ( IN DICTIONARY AND ZERO IT )

: OTHERCASE  DUP 8000 SWAP ! ; IMMEDIATE  ( SET OTHERCASE BIT )
                                                ( IN CASE-INDEX )

: DONE  DUP HERE SWAP - 2 / 1 -     ( CALC. CNT FOR CASE-INDEX )
   SWAP DUP @     ( GET THE CASE-INDEX TO TEST FOR OTHERCASE )
   ROT DUP 0=                       ( TRUE IF NO ITEMS IN LIST )
     IF DROP DROP 0                   ( SET CASE-INDEX TO ZERO )
     ELSE SWAP                  ( TRUE IF OTHERCASE SPECIFIED )
       IF 1 - 8000 OR THEN       ( = -1 AND OTHERCASE BIT SET )
     THEN SWAP ! ; IMMEDIATE  DECIMAL ;S  ( STORE CASE-INDEX )
```

This listing is from a 6809 version of FORTH.

CASE

n CASE m0 m1 ... mi DONE

CASE is used as a structured construction where n = 0 to i and m0 m1 ... mi represent a list of word names with the list being terminated by the word DONE.

When the definition containing the case construction is executed, module mn will execute, then execution will continue after the DONE. If n is not in the range 0 to i, execution continues after the DONE.

Alternative CASE usage with OTHERCASE

n CASE m0 m1 ... mi OTHERCASE mx DONE

When the definition containing the case construction is executed, module mn will execute if n is in the range 0 to i; then execution will continue after the DONE. If n is not in the range 0 to i, module mx will execute and then execution will continue after the DONE.

Only executable modules should be used in the case list; literals and compiler words, especially:

CASE OF ELSE THEN BEGIN END BUILDS DOES e

Should NOT be used.

OTHERCASE

Used in conjunction with CASE word for out of range conditions. See CASE usage.

DONE

CASE word terminator. See CASE usage.

I!

n ---

Replaces the interpreter pointer with the top stack item (n).

(CASE)

n ---

The execution time portion of the CASE word.

<<

n1 n2 --- f

Unsigned 16 bit less than.

Example of CASE usage

```
: TXX CASE TX1 TX2 TX3 DONE ;
```

If TXX is executed, then execution will continue as follows based on the value on the stack.

STACK VALUE	EXECUTE
0	TX1
1	TX2
2	TX3

Execution then continues after the DONE. If the stack value was not 0, 1 or 2 then execution continues after the DONE.

Examples of CASE usage with OTHERCASE.

```
: MH2 TE1 CASE ENQ VOICE SYNC NULL OTHERCASE EM3 DONE ;
: MH1 CASE NULL FIFO TIME XMIT-MSG OTHERCASE MH2 DONE CLEANUP ;
```

If MH1 is executed, then execution will continue as follows based on the value on the stack.

STACK VALUE	EXECUTE
0	NULL
1	FIFO
2	TIME
3	XMIT-MSG
Any Other Value	MH2

Execution then continues after the DONE, in this instance CLEANUP.

MH2 illustrates the nesting capability of the CASE word.

This form of CASE conforms with the unwritten rule of FORTH to keep it simple and basic. The user needs to remember only three words, CASE, OTHERCASE and DONE to construct simple to complex forms of the structured CASE. The CASE in providing automatic limit checking and out of range recovery elliminates the need for user limit testing of the parameters. This out of range checking capability does slow the execution speed slightly, but it was felt that the added capability was worth the slight loss of speed.

Bill Busler
Odessa, Florida 33556

Wayne Witt
Tampa, Florida 33615

Judge's Comments - The run-time word (CASE) seems much too long for the job it does. This is partly because the out-of-range case is handled by a special construction. Nevertheless, the code could be reorganized or factored. Also, pushing the DONE address back on the return stack at the end of (CASE) would eliminate the need for I! and make the package more portable.

The GODO ... THEN construction in Kitt Peak FORTH accomplishes all the same functions much more efficiently.

COME TO FIG CONVENTION
NOVEMBER 29

THE KITT PEAK GODO CONSTRUCT

By David Kilbridge

The GODO construct, as specified in the glossary of the Kitt Peak FORTH Primer, is a type of CASE statement. An index on the stack is truncated to fall within a contiguous range and used to select a word from an in-line execution vector. I present here a very simple implementation in fig-FORTH.

As an example of usage, here is a word which accepts a 0 or 1 from the terminal and selects the corresponding disk drive, and rings the bell if any other key is pressed.

```
: GET-DRIVE   ." DISK DRIVE? "
   KEY  2F -
   GODO  BELL DR0 DR1 BELL  THEN  ;
```

The necessary source definitions are

```
: (GODO)   2*
   0  MAX   R  @  4  -  MIN
   R>  DUP  DUP  @  +  >R
   +  2+  @  EXECUTE  ;

: GODO
   COMPILE  (GODO)
   HERE  0  ,  2  ;
   IMMEDIATE
```

How it works: GODO compiles (GODO) and leaves space for a branch offset to be calculated by THEN. The address of the cell and an error-checking flag are left on the stack. At run time (GODO) doubles the index on the stack and truncates it both above and below so that the reference executed will always be chosen from the list provided. Then (GODO) uses the branch offset to step its return address over the reference list and finally executes the selected reference.

Glossary:

```
GODO     --- addr n (compile-time)   P,C
(GODO) n ---        (run-time)
```

Used in the sequence

```
... GODO  R0  R1 ... Rn  THEN ...
```

At run-time, GODO selects execution based on a signed integer index. If the index is <=0 then R0 is executed; if =1 then R1 is executed; ... if >=n then Rn is executed. After executing the selected reference, execution resumes after THEN.

Discussion: The GODO construct provides a basic contiguous-range type of CASE statement requiring very little supporting code. The compile-time word is simple because most of the work is done by THEN. The run-time word is simple because truncating the index allows out-of-range cases to be handled just like in-range cases.

If other means are used to insure that the index is always within range, the "catch-all" references R0 and/or Rn can be omitted. However, there is still the time overhead needed to truncate the index (unless (GODO) is recompiled without the second line of its definition).

The principal limitation of this construct is that only single words can be referenced. This prevents direct nesting of GODO's. However, one can nest by defining the inner GODO as a separate word and referencing it in the outer GODO. By letting Ro and/or Rn be such references, several noncontiguous ranges can be covered.

Kitt Peak PRIMER available from FIG for $20.00 in US and $25.00 Overseas.

FIG NORTHERN CALIFORNIA MONTHLY MEETING REPORT

26 April 80

The FORML session consisted of three presentations covering FORTH File proposals. John James and John Cassady discussed Directories consisting of bit maps named FileControlBlock (FCB) wherein allocation of strings of blocks (Files) were managed. Particulars of bitmap manipulation at the Buffer, Block and Disk (file and volume) levels were explicated. Some other concepts included user transparency, hierarchy of directories, commands, security and integrity. Kim Harris described Record types and management within a File and gave examples of FORTH, Inc. styled I/O at the Field level. The pros and cons of the various approaches will be debated at the next meeting where also String manipulation will be discussed. Attendees were requested to prepare written proposals of anticipated requirements and arguments for and against the different approaches. Though not a tutorial, the FORML session was very instructive.

The April Northern California FIG meeting consisted of a presentation by Jim Brick (of M&B Design) of a polyFORTH bootup under CP/M. Jim described the application requirements that produced the need and the technique he used to develop this bootup package sold by FORTH, Inc. He demonstrated the hybrid package on a TRS-80 with I/O accessories which allowed 8" disks and remapping of the TRS-80 memory for polyFORTH-CP/M compatability.

Bill Ragsdale initiated a tutorial on overflow correction which spontaneously escalated into a discussion on error signals, repair and recovery. Kim Harris, Lafarr Stuart and Dave Boulton described their respective approaches to dealing with errors. Bill elaborated the "Utrecht approach" to error signaling and recovery and noted two lessons learned: high level words can define error recovery and the return stack can be usefully unthreaded. He congratulated our Dutch colleagues for their imaginative applications of "tricks" garnished from other computer languages.

Henry Laxen was congratulated for his excellent article on FORTH in the 80 April 28 issue of INFOWORLD.

Kim Harris announced his FORTHcoming course on FORTH programming at Humbolt State University (80 July 21-25) and also reported on a talk he delivered earlier this month at the Asilomar I.E.E.E. conference on megatransistor chips.

...HANDOUTS provided at the meeting included:

-polyFORTH-CP/M (Brick)

-INFOWORLD reprint (Laxen)

-TIC-TAC-TOE (in FORTH, of course) (George Flammer)

-overflow correction (Ragsdale)

-Match CPM for 8080 figFORTH (anon)

-Double number support (Ragsdale)

-String match for Editor (Peter Midnight)

;s Jay Melvin

Publisher's Note:

Come on, you other FIGGERS, send in reports on your meetings. We'll publish them.

FIG NORTHERN CALIFORNIA MONTHLY MEETING REPORT

24 May 80

FORML Session -

Kim Harris directed a review of last month's session to compare and contrast file systems presented by:

1. John James
2. John Cassady
3. Kim Harris (FORTH, Inc. system)

The most striking difference between the three file systems was that FORTH, Inc.'s did not utilize a bit map in the directory which would allow for a distinction between physical and logical files. The bit map implemented in James' and Cassady's systems provide for easier file manipulation.

FIG Meeting -

Bill Ragsdale opened the meeting by introducing guests Ed Murray from the University of South Africa and Don Colburn who is marketing a FORTH Teaching Tutorial to be configured for various machines.

The meeting was devoted to a two fold tutorial where Kim Harris explained FORTH tools ranging from NUMERIC output and base conversion to test interpretation. I/O formating examples included the definition of HOLD, ASCII and PAD. These "tools" were applied in a temperature conversion program. Bill Ragsdale followed with a presentation on problem solving techniques using the task of printing Morse (dits/dahs) characters to the screen in response to text input. Top down techniques were delineated by listing the subtasks and writing code then testing each module.

John Draper described CAP'N Software's Version 1.7 FORTH for the Apple; the system was up and running for demonstration. Ragsdale notified us that Computer magazine wants articles for a FORTH issue next year and that Byte's August issue will have a Robert Tinney cover displaying three blocks in a field of stars, each block containing a word (2*, DUP, +) and threaded together by a ribbon terminating in a space needle.

Handouts included: Kim's tool kit, Bill's Morse Code worksheet (a blank page!), John's Version 1.7 brochure, and Benchmark by DRC for measuring FORTH execution speeds on CRAY-1 through micros. Also, a floating point package by NHC, a paper on file word concepts by Jim Berkey and the HomeBrew Computer Club's newsletter by (ed.) Bill Reiling were available.

;s Jay Melvin

FORTH Interest Group Meetings

Northern California 4th Saturday	FIG Monthly Meeting, 1:00 p.m., at Liberty House Department Store, Hayward, CA. FORML Workshop at 10:00 a.m.
Massachusetts 3rd Wednesday	MMSFORTH Users Group, 7:00 p.m., Cochituate, MA. Call Dick Miller at (617) 653-6136 for site.
San Diego Thursdays	FIG Meeting, 12:00 noon. Call Guy Kelly at (714) 268-3100 x 4784 for site.
Seattle Various times	Contact Chuck Pliske or Dwight Vandenburg at (206) 542-8370.
Potomac Various times	Contact Paul van der Eijk at (703) 354-7443 or Joel Shprentz at (703) 437-9218.
Texas Various times	Contact Jeff Lewis at (713) 729-3320 or John Earls at (214) 661-2928 or Dwayne Gustaus at (817) 387-6976. John Hastings (512) 835-1918.
Arizona Various times	Contact Dick Wilson at (602) 277-6611 x 3257.
Oregon Various times	Contact Ed Krammerer at (503) 644-2688.
New York Various times	Contact Tom Jung at (212) 746-4062.
Detroit Various times	Contact Dean Vieau at (313) 493-5105.
Japan Various times	Contact Mr. Okada, President, ASR Corp. Int'l, 3-15-8, Nishi-Shimbashi Minato-ku, Tokyo, Japan.

Publisher's Note:

Please send notes (and reports) about your meetings.

----HELP WANTED----

BUSINESS SYSTEMS IN FORTH

We need two good FORTH programmers.

You should have solid FORTH experience, a year or two, and be generally competent in Computer Science.

We are building an exciting range of business application systems using FORTH - the advantages are obvious! - and our approach is unique. We'll have a range of configurations - single and multi-processor, both Winchester and large fixed disks and color graphics screens.

Ideally you'll live in Orange County - be attracted by a small, quality team - and like to grab your own projects with a strong sense of self management - we haven't got the time or the inclination to be overbearing.

Please send brief description of your background to:

The Software Development Director
4861 McKay Circle
Anaheim, CA 92807

and let us know why you think you'd like to work with us.

CALL FOR PAPERS

FORML CONFERENCE

(FORTH Modification Laboratory)

Papers are requested for a three day technical workshop to be held November 26-28, 1980 at the Asilomar Conference Grounds in Pacific Grove, California (on the Monterey Peninsula). The purpose of the workshop is to discuss advanced technical topics related to FORTH implementation, language and application. Papers on any of the following or related topics are requested for presentation and discussion:

1. Programming methodology
   - problem analysis and design
   - implementation style
   - development team management
   - documentation
   - debugging
2. Virtual machine implementation
   - arithmetic
   - address enlargement
   - position independent object code
   - metaFORTH
3. Concurrency
   - resource management
   - scheduling
   - intertask communication and control
   - integrity, privacy and protection
4. Language and compiler
   - typing and generic operations
   - data and control structures
   - optimization
5. Applications
   - file systems
   - string handling
   - text editing
   - graphics
6. Standardization
   - Review and discussion of 79-STANDARD
   - Input for the Standards Team

FORML is an organization (sponsored by the FORTH Interest Group) which promotes the exchange of ideas on the use, modification and extension of the FORTH approach to systems development. This will be an advanced technical workshop; no introductory tutorials will be held.

Abstracts of papers must be received by October 1, 1980 for inclusion in the conference program. Complete papers must be received by November 1, 1980 to be included in the conference proceedings. Send both abstracts and completed papers to:

FORML Conference
P. O. Box 51351
Palo Alto, CA 94303

----HELP WANTED----

TITLE: Product Support Programmer

DUTIES: Responsible for maintaining existing list of software products, including the polyFORTH Operating System and Programming Language, file management options, math options and utilities and their documentation, and providing technical support to customers of these products.

Requirements for candidates:

1. Good familiarity with FORTH—preferably through one complete target-compiled application.

2. Good assembler level programming skills.

3. Assembler level familiarity with the 8080 and PDP/LSI-11 processors and preferably some of these: 8086, M6800, CDP1802, NOVA, IBM Series I, TI990.

4. Excellent communications skills--both oral and written; ability to work well with customers.

5. Excellent organizational ability.

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

FORML CONFERENCE

(FORTH Modification Laboratory)

November 26-28. 1980 at the Asilomar Conference Grounds, Pacific Beach, California. A three day advanced technical workshop for the discussion of topics related to FORTH implementation, language and application. No introductory tutorials will be held.

FORML is an organization (sponsored by the FORTH Interest Group) which promotes the exchange of ideas on the use, modification and extension of the FORTH approach to systems development.

Asilomar is a comfortable, rustic resort located on the Pacific Ocean near Monterey in Northern California. Attendees are urged to bring family members to Asilomar as they will enjoy the area and Thanksgiving dinner. Costs are very reasonable, especially for families, and include room (double occupancy) and meals.

Attendees and/or participants	$100.00 (includes conference registration and materials)
Non-conference guest (wife and/or husband, friend, and children 12 or over)	$ 75.00
Children 11 or younger	$ 50.00

Send request for registration and list of guests by October 15th with a check to:

FORML Conference
P.O. Box 51351
Palo Alto, CA 94303

NATIONAL CONVENTION

FORTH Interest Group

November 29, 1980 at the Villa Hotel, San Mateo, California, 8:30 a.m. - 4:30 p.m. for exhibits and papers; 6:00 p.m. cocktails; 7:30 p.m. for dinner (with speaker). This one day convention will include presentations, workshops, hands-on equipment and a number of vendor exhibits. An evening dinner will include a talk by one of the foremost authorities on FORTH (more about the speaker in a later release).

Pre-registration for the convention is available for $4.00.

Pre-registration for the dinner and speech is required by October 15th at $15.00.

Vendors may contact FIG about the cost and availability of booth and table space.

To pre-register or for more information write:

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

Vendors may contact Roy Martens at (415) 962-8653 for details about exhibiting.

Room arrangements can also be made through FIG.

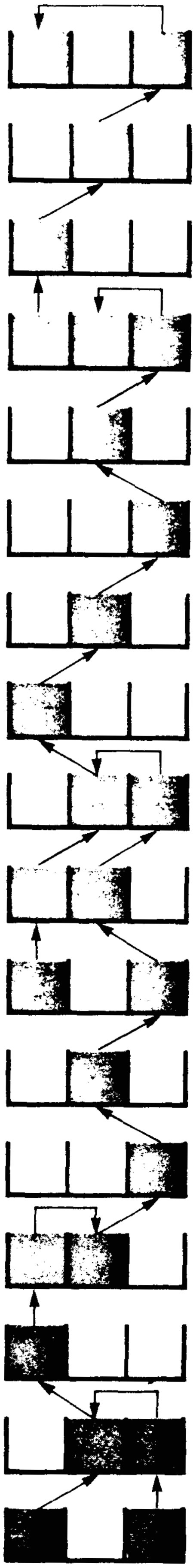
************FLASH LATE NEWS************

FIG NATIONAL CONVENTION BANQUET SPEAKER

ALAN TAYLOR

Author of The Taylor Report for Computer World. 30 years in computer field.

*********MAKE YOUR RESERVATION*********



FORTH DIMENSIONS

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

Volume II
Number 4
Price $2.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No. 4 November/December 1980

Publisher Roy C. Martens

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL 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 $12.00 per year ($15.00 overseas). 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 for 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, although our membership of 2,000 is worldwide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

PUBLISHER'S COLUMN

We're deep into the planning and arrangements for the FIG Convention and the FORML Conference. If you haven't made your reservations, call right away, we might be able to get you into the FORML Conference or the Convention Banquet. Plan on coming to the Convention anyway. Remember the dates and places are:

FORML Conference, November 26, 27, & 28
Asilomar, CA

FIG Convention, November 29
Villa Hotel, San Mateo, CA

The other big news! FORTH-79 STANDARD is available!!! Call (415) 962-8653 or send in your order, today! $10.00!

Many publications are printing information about FORTH. We don't get them all, so please send in copies so we can thank the editors and add to our collection.

FIG had a booth at the Mini/Micro show and much interest was generated among attendees which carried over into a number of manufacturers that were exhibiting.

Membership is fast approaching 2,000. We now have members all over the world including the People's Republic of China and Yugoslavia. See the listings of meetings for information about how you can form a FIG chapter. Just a few easy steps and you'll have a time and place to share information.

Look forward to seeing everyone at the FORML Conference and the FIG Convention.

Roy Martens

BALANCED TREE DELETION IN FASL

Douglas H. Currie, Jr.
Nashua, NH

Abstract

FASL (Functional Automation Systems Language) is a derivative of FORTH containing significant modifications. This paper discusses one of these, the FASL tree, an implementation of the AVL (height balanced) tree. FASL trees are a data type of the language, and are used in the implementation of the dictionary. An algorithm for deletion in FASL trees is presented, as well as a FASL program to implement the algorithm.

Key Words and Phrases

deletion, height-balanced trees, binary trees, search trees, FORTH.

CR Categories

3.7, 4.10, 4.20, 4.34, 5.25, 5.31

Introduction to Height-Balanced Trees

The use of balanced trees has become almost commonplace in data base management, and is seeing limited use in symbol tables. Many systems would benefit from the use of balanced trees, but their designers could not afford the time to develop the algorithms. A case in point is the extensive use of hashing in "high-speed" microcomputer assemblers. Hashing techniques have significantly improved the performance of many assemblers, but analysis of these routines shows a best case performance on the order of several milliseconds (due to the inefficiency of division, or pseudo-random number generation on microprocessors). FASL trees, on the other hand, have a guaranteed worst case performance of far less than a millisecond even in fairly large (over five hundred node) trees.

In FUNCTIONAL* systems, FASL trees are used in a line editor, data storage directories, FACT (a truth table compiler), message routing tables, microcomputer assemblers, as well as the FASL dictionary. A general purpose microassembler uses a balanced tree (fields) of balanced trees (contents) to describe the target microinstruction. The use of multiple trees allows identical keys in different contexts (e.g., label names and macro names).

The height-balanced tree was first proposed by two Russian mathematicians, G. M. Adel'son-Vel'skiy and E. M. Landis in 1962 (hence AVL tree). The idea is to maintain a binary tree so that the height of the subtrees at any node differ by at most one. The technique incurs a penalty of only two extra bits per node (FASL uses an 8-bit byte), and makes it possible to search for, insert, or delete a node with a worst case of O(log N) operations (where N is the number of nodes).

Introduction to FASL Trees

Algorithms for search and insertion in AVL trees are presented by Knuth (The Art of Computer Programming, Vol. 3, Section 6.2.3); these two algorithms were implemented in machine code and (along with Indirect Threaded Code) became the basis for FASL. The deletion algorithm was not implemented at this time for two primary reasons: Knuth didn't give it, FASL didn't "need" it. Deletions occur much more rarely than insertions or searches; FASL lived for over a year with no delete operation.

---

*Functional Automation Gould Inc.
3 Graham Drive
Nashua, NH 03060

For example, when a file was deleted from a FASL directory, the entire directory was reconstructed without the "deleted" node. The time penalty incurred was not significant because directories are small (for FASL trees), and had to be copied anyway to be sent to the disk. (FASL lives in a message enviroment. The disk is in another Cyblok*).

After an overview of FASL trees and their use, the remainder of this paper will deal with the development of a FASL tree deletion program in FASL. For an introduction to binary search trees, see Knuth (The Art of Computer Programming, Vol. 3).

FASL trees are composed of a number of sixteen byte nodes (see Figure 1). The tree is identified with the address of its head node. From the head node we may find the root node, and thus the entire tree. The head node contains a pointer to its root node, a pointer to its available nodes list, and an integer which is the tree's height.

All nodes other than the head node contain an eight byte key, a left link, a right link, a one byte balance factor, and three uncommitted bytes. The key is used to access the node. Given a key, the search routine compares it to the key at the root node. If it is less, the search continues with the node identified (pointed to) by the left link. If it is greater, the search continues with the node identified by the right link. The search terminates when it matches the key (success), or reaches a null link (failure). The null link is represented by zero. The balance factor is the height of the right subtree minus the height of the left subtree. The insertion routine always leaves the tree balanced, i.e., the balance factor is always minus one, zero, or plus one.

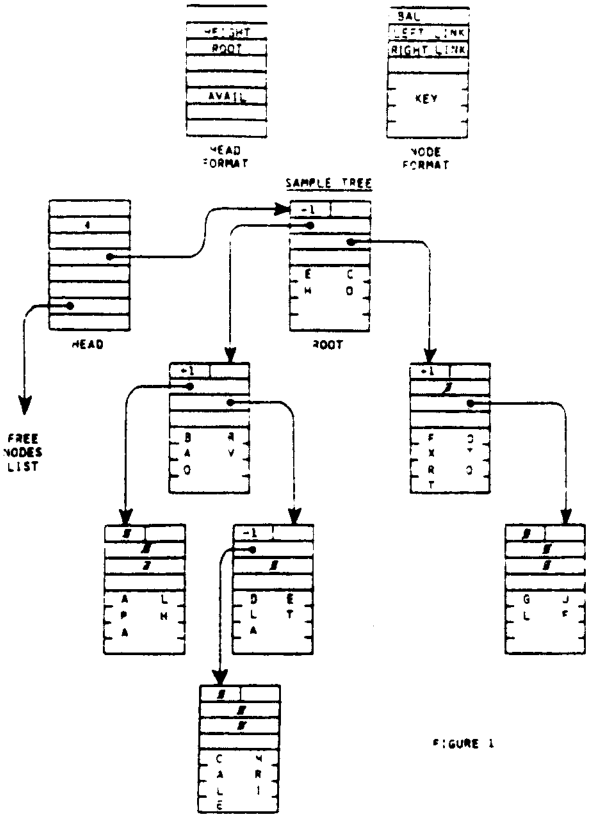


FIGURE 1

The insertion routine obtains new nodes from the free nodes list. This list is simply a number of nodes linked with their right links. A null right link indicates the end of the free nodes list. When the insertion routine needs a free node, it obtains its address from the free nodes list pointer in the head node, and replaces it with the right link of that node. If the free nodes list pointer is null, then the tree is full.

The technique used by the insertion routine to maintain tree balance is essentially the same as for deletion. Basically, four cases arise in insertion when the tree must be rebalanced: single or double rotation, left or right. The discussion is postponed until the section on deletion.

*Cyblok is a registered trademark of Functional Automation/Gould Inc.

To get a feeling for the efficiency of FASL trees, consider a dictionary of five hundred nodes. If this dictionary was stored as a linked list, a worst case access time of five hundred compares would be incurred, with an average access time of two hundred fifty compares. Stored as a FASL tree, this dictionary has a worst case access time of nine compares, an average of eight. The numbers become even more convincing as the dictionary grows in size.

FASL Tree Operations

FASL provides operations for creating trees, inserting and searching for nodes, and accessing the uncommitted data in a node. For example, the FASL text

```
100  TREE  SYMBOLS
```

creates a tree named SYMBOLS with two hundred fifty-six available nodes (the radix is hexadecimal). Assuming there is a string of text in an area named PAD which is to be used as a key to access the tree,

```
PAD  SYMBOLS  LEAF
```

inserts a node in the tree SYMBOLS with this key. LEAF leaves a boolean flag on the stack to indicate success or failure, and if successful leaves the address of the new node on the stack under the boolean.

Usually, new nodes are initialized with some data. The following FASL text will insert a node with the key in PAD (as above), and initialize its uncommitted bytes with constants:

```
12  3456  PAD  SYMBOLS  LEAF
    IF  F#!
    ELSE  DROP2  FI
```

Later, the data may be retrieved onto the stack as follows:

```
PAD  SYMBOLS  FIND
    IF  F#@
    ELSE  FAIL  FAIL  FI
```

If the string in PAD is the same as was used in the preceding example to insert the node, then the data retrieved will be 12 3456. If another string is in PAD, then the data retrieved will be 00 0000, unless a node has been inserted with this string as a key, in which case the data associated with this node will be retrieved.

From the example, it should be clear how to use the FASL trees for a symbol table for an assembler. Text is read to PAD until a delimiter, and then inserted in the tree. In the case of labels, the node would be initialized with the current pseudoPC, and a flag byte to indicate "label." If the inserted text was a macro name, the node might be initialized with a pointer to the macro text and a flag byte to indicate "macro." Alternatively, separate trees may be created so that identical keys may be used as macro and label names. Later, when a label or macro is used, it may be looked up in the tree to find its corresponding values.

The TREE operation allocates space for the tree in the FASL Global Area (where code for colon-words is placed). Another operation, TREEINIT, is provided to initialize trees in space that the FASL user has allocated (e.g., in FUNCTIONAL Cybloks there is a minimum of 256K bytes of "Public Memory" which is accessed through "Windows," and is not part of the FASL Global Area). The TREEINIT operation is often used in the Local Area (space allocated on the Return Stack) or in Public Memory.

The Deletion Algorithm for FASL Trees

A deletion algorithm for binary trees, and the steps required to adapt this algorithm to balanced trees are provided by Knuth (The Art of Computer Programming, Vol. 3, Sections 6.2.2 and 6.2.3). The details of the balanced tree deletion algorithm are presented here, but first a review of binary tree deletion.

Deleting a node from a binary tree may be decomposed into four cases (see Figure 2). Call this node "X". In the first two cases one of the links of X is null, the other link is a "don't care" (i.e., a pointer or null). In both cases the other link simply replaces the link pointing to X. In case three the right son of X has a null left link. In this case the left link of X replaces the left link of its right son, and the right link of X replaces the link pointing to X. In case four the symmetric successor of X must be found. This is done by following left links starting with the right son of X until a null link is encountered. The left link of the father of the symmetric successor is replaced by the right link of the symmetric successor. The left and right links of the symmetric successor are replaced by the respective links of X, and the link which points to X is replaced by a pointer to the symmetric successor.

In all cases the essential left-to-right order of the nodes is preserved. The deleted node is inserted in the free nodes list, and the algorithm terminates.

All that is required (!) to adapt this algorithm to balanced trees is to insure that the balance is maintained after the deletion. An important observation is that the effect of deletion on the binary tree is to reduce the length of a single path through the tree by one.

This path begins at the head, and ends in cases one and two with the node which re- placed X (i.e., the node which is pointed to by the link which used to point to X). In cases three and four the path ends with the node which used to be the right son of the symmetric successor of X. (Note that the ending node may actually be null.)

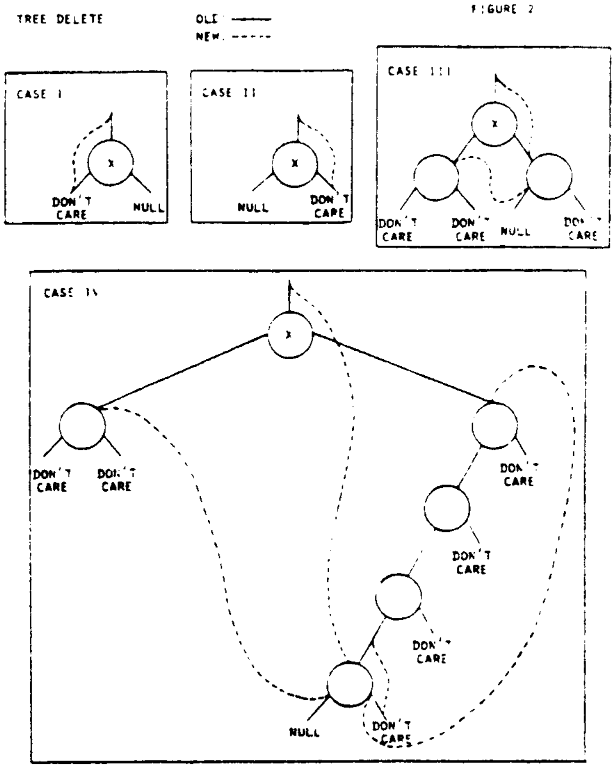


FIGURE 2

The path may be represented as a list of pairs

(N.0 , f.0) (N.1 , f.1)
... (N.i , f.i)

where each N.j is a node address, and each f.j is a direction (-1 left, +1 right). N.0 is the head node, f.0 is the +1 (since the "right link" of the head node points to the root). The pair (N.i , f.i) is the end node minus one, and identifies the end node of the path (which, again, may be null). Rebalancing may be required at each node in the path, starting with node (N.i , f.i), working backwards. This is in contrast to insertion where rebalancing is required for, at most, one node.

Adapting the deletion algorithm for binary trees to balanced trees requires that as the tree is searched for the node to be deleted (and for its symmetric successor in cases three and four), a list of pairs describing the path is created. Once the node is deleted, nodes are rebalanced back along the path until a termination condition is reached. .

The path is constructed on an auxiliary stack. The operations "Push(x,y)" to push a pair, "Pop(x,y)" to pop a pair, and "Top(x,y)" to read the top pair without popping are used, as well as the capability of saving and restoring the path stack pointer.

Using the notation "Link(-1 , M)" for left link of node M, "Link(1 , M)" for right link of node M, "Bal(M)" for the balance factor of node M, and "Key(M)" for the key of node M, the following is a detailed algorithm for deleting the node with key K in a balanced tree.

```
(1)  Initialize local path stack.
     Push(HEAD , +1).
     Set X to Link(+1 , HEAD).

(2)  If K is less than Key(X), go to (3) moving left.
     If K is greater than Key(X), go to (4) moving right.
     Otherwise go to (5), key is found.

(3)  If Link(-1 , X) is 0, go to (11), key is not in tree.
     Otherwise Push (X , -1), set X to Link(-1 , X), and go to (2), keep searching.

(4)  If Link(1 , X) is 0, go to (11) key is not in tree.
     Otherwise Push(X , 1), set X to Link(1 , X), and go to (2), keep searching.

(5)  There are four cases:

   (5a)  Link(1 , X) = 0 ;
         Top(N.k , f.k).
         Set Link(f.k , N.k) to Link(-1 , X).
         Go to (7) to rebalance.

   (5b)  Link(-1 , X) = 0 ;
         Top(N.k , f.k).
         Set Link(f.k , N.k) to Link(1 , X).
         Go to (7) to rebalance.

   (5c)  Link(-1 , Link(1 , X)) = 0 ;
         Top(N.k , f.k).
         Set Link(-1 , Link(1 , X)) to Link(-1 , X).
         Set Link(f.k , N.k) to Link(1 , X).
         Set Bal(Link(1 , X)) to Bal(X).
         Go to (7) to rebalance.

   (5d)  Otherwise ; Push(X , 1), set Z to Link(1 , X).
         Save path stack pointer in PSP.
         Go to (6) to find symmetric successor.

(6)  Push (Z , -1).
     Set Z to Link(-1 , Z).
     Repeat this step until Link(-1 , Z) = 0.
     Finally, Top(N.k , f.k).
     Set Link(-1 , N.k) to Link(1 , Z).
     Set Link(-1 , Z) to Link(-1 , X).
     Set Link(1 , Z) to Link(1 , X).
     Now swap PSP and the path stack pointer.
     Pop(N.k , f.k) ,
     Top(N.k , f.k), Push(Z , 1), substituting the symmetric successor for the deleted node on the path stack.
     Swap PSP and the path stack pointer again to restore.
     Set Link(f.k , N.k) to Z.
     Set Bal(Z) to Bal(X).
     Go to (7) to rebalance.
```

(7) Insert X into the free nodes list.

The algorithm proceeds as follows beginning with the last pair of the path:

(8) Pop(N.k , f.k).
If N.k = HEAD, set Height(HEAD) to Height(HEAD)-1 decreasing the height of the tree, and go to (11) terminating the algorithm.
Otherwise go to (9).

(9) There are three cases based on the balance factor:

(9a) Bal(N.k) = 0 ; Set Bal(N.k) to -f.k, and go to (11) terminating the algorithm.

(9b) Bal(N.k) = f.k ; Set Bal(N.k) to 0, and go to (8) taking one more step back along the path.

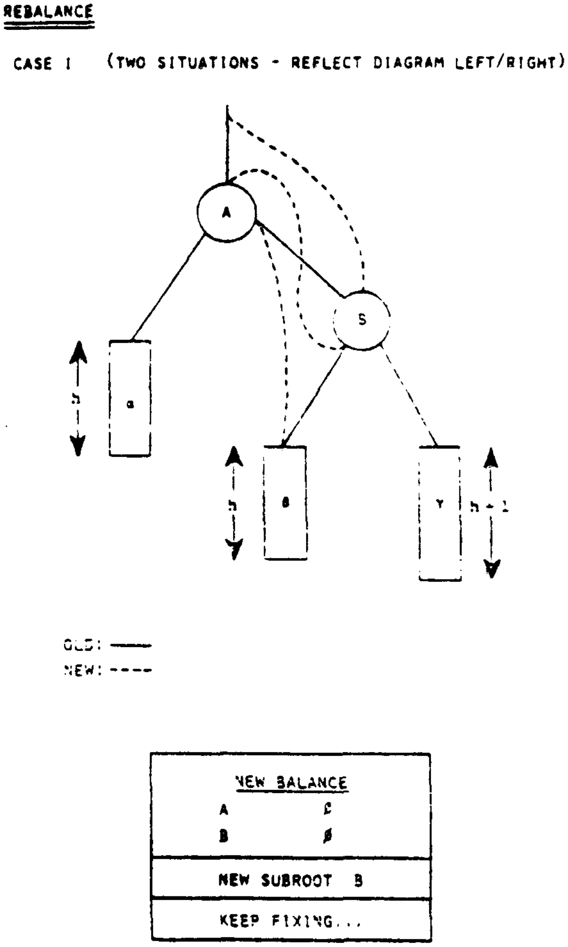
(9c) Bal(N.k) = -f.k ;
Rebalancing is required, go to (10).

(10) There are again three cases. (Referring to Figures 3, 4, and 5, A is N.k, α is the subtree containing the path the algorithm has been following, B is the node pointed to by the <u>opposite</u> link from the link which points to α, Link(-f.k , N.k)):

(10a) Bal(A) = Bal(B) (Figure 3);
Set Bal(A) and Bal(B) to 0.
(single rotation) -
Set Link(-f.k , A) to Link(f.k , B).
Set Link(f.k , B) to A.
Top(N.k , f.k), set Link(f.k , N.k) to B.
Go to (8) taking one more step back along the path.

(10b) Bal(A) = -Bal(B) (Figure 4); If Bal(X) = Bal(A), then set Bal(A) to -Bal(X) and Bal(B) to 0.
Otherwise set Bal(A) to 0 and Bal(B) to -Bal(X).
Set Bal(X) to 0.
(double rotation) -
Set Link(-f.k , A) to Link(f.k , X).
Set Link(f.k , X) to A.
Set Link(-f.k , B) to Link(-f.k , X).
Set Link(-f.k , X) to B.
Top(N.k , f.k), set Link(f.k , N.k) to X.
Go to (8) taking one more step back along the path.

FIGURE 3



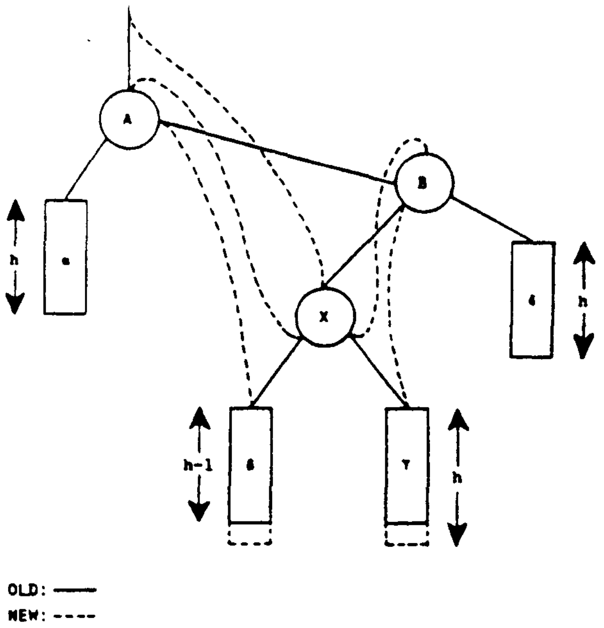
(10c) Bal(B) = 0 (Figure 5);
Set Bal(B) to -Bal(A).
(single rotation) -
Set Link(-f.k , A) to Link(f.k , B).
Set Link(f.k , B) to A.
Top(N.k , f.k), set Link(f.k , N.k) to B.
Go to (11) terminating the algorithm.

(11) Deallocate path stack. Done!

FIGURE 4

REBALANCE

CASE II (TWO SITUATIONS - REFLECT DIAGRAM LEFT/RIGHT)

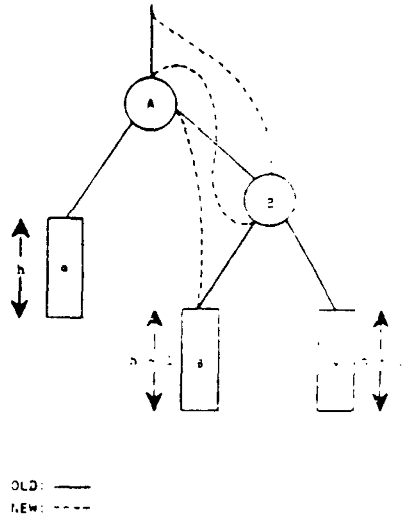


NEW BALANCE	BAL(X) = BAL (A)	OTHERWISE
A	-BAL(X)	0
B	0	-BAL(X)
X	0	0
NEW SUBROOT	X	
KEEP FIXING...		

FIGURE 5

REBALANCE

CASE III (TWO SITUATIONS - REFLECT DIAGRAM LEFT/RIGHT)



NEW BALANCE	
A	BAL(A)
B	-BAL(A)
NEW SUBROOT	B
DONE!	

Implementing the Algorithm in FASL

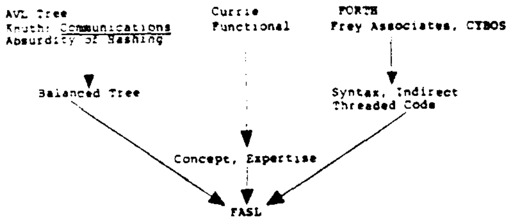
A FASL program to implement the balanced tree deletion algorithm is relatively straightforward (see the listing below). Some preliminary colon-words are defined to access the links, and to access a Local Stack. RCRUMB and LCRUMB are defined (in commemoration of Hansel and Gretel) for adding pairs to the path stack; then colon words for the three cases encountered in rebalancing are defined.

The main colon-word, DROPLEAF, takes stringname and treename parameters just like LEAF and FIND, but leaves no return values since it is always successful. The PROC... ENDPROC pair allocate and deallocate a Local Data Area for the path stack and associated variables. For the most part, DROPLEAF follows the deletion algorithm presented. Nested IF statements are used to evaluate the case constructs. The string compare in the first (search) WHILE loop tests for less-than directly, and examines FASL Registers (W0, W1) to resolve the trichotomy. (This is an efficiency measure, and has to do with the fact that there is not guaranteed to be a string delimiter in the node's key.)

Empirical tests show that DROPLEAF runs in the 50 to 100 millisecond range for trees with about 500 nodes. For comparison, LEAF runs in the 0.1 to 1 millisecond range on the same trees. The large difference between these runtimes results from the fact that LEAF is highly optimized machine code, only requires one rotation maximum, and does not require a path stack. As previously mentioned, DROPLEAF is used very infrequently, and there has been no incentive to implement it in machine code.

```
( HEIGHT BALANCED )
( TREE DELETE )
( 17Mar80 )

( LOCAL DATA AREA )
( OFFSET )
( ------ )
(
    2   saved path stack pointer
    4   path stack pointer
    6   address of link to node to be deleted
    8   start of path stack
    .
    .
    .
    30  end of path stack + 1
)

( 1,1 )
: LLNK@ 2 + @ ;
: RLNK@ 4 + @ ;
( 2,0 )
: LLNK! 2 + ! ;
: RLNK! 4 + ! ;

( 1,0 )
: PUSH 4 'D @ ! 2 4 'D +! ;
( 0,1 )
: POP OFFFE 4 'D +! 4 'D @ @ ;
( 1,1 )
: RCRUMB DUP PUSH OFFFF PUSH ;
: LCRUMB DUP PUSH SUCCEED PUSH :

( 3,2 )
: SINGLROT OVER2 LTZ?
    IF DUP RLNK@ OVER2 LLNK!
      SWAP OVER RLNK!
    ELSE DUP LLNK@ OVER2 RLNK!
      SWAP OVER LLNK!
    FI ;
```

```
: ROTCASE1 FAIL OVER C! FAIL OVER2 C!
    SINGLROT SWAPDROP FAIL SWAP ;

: ROTCASE3 OVER C@ NEG OVER C!
    SINGLROT ;

: ROTCASE2 OVER2 OVER2 OVER2 OVER2 - 3 + @
    SINGLROT
    SWAP NEG SWAP OVER2 SWAP
    SINGLROT SWAPDROP
    OVER2 C@ OVER C@ -
      IF DUP C@ NEG SROT C! FAIL SROT C!
      ELSE FAIL SROT C! DUP C@ NEG SROT C! FI
    FAIL OVER C!
    SWAPDROP FAIL SWAP;

: MOVEZR + DUP 6 'D ! @ ;

( 2,0 )
( <sname> <tname> )
: DROPLEAF
    30 PROC
    8 'D 4 'D !
    SWAP OVER
    RCRUMB
    4 MOVELZR
    WHILE DUP
      IF OVER OVER 8 + SLT? DUP
        IF OVER 10 + W0 @ -
        ELSE W1 @ 1 - C@ FI
      ELSE FAIL FAIL FI
    CONTINUE
      IF LCRUMB 2
      ELSE RCRUMB 4 FI
      MOVELZR
    WHILEND
    DROP
    SWAPDROP
    DUP
      IF DUP RLNK@
        IF DUP LLNK@
          IF DUP RLNK@ DUP LLNK@
            IF 4 'D @ 2 'D ! RCRUMB
              DUP
              REPEAT LCRUMB SWAPDROP DUP LLNK@ DUP LLNK@ ZERO?
              UNTIL
              OVER2 LLNK@ OVER LLNK!
              DUP RLNK@ OVER2 LLNK!
              OVER2 RLNK@ OVER RLNK!
              SWAPDROP
              DUP 2 'D @ :
            ELSE OVER LLNK@ OVER LLNK!
              RCRUMB
            FI
            OVER C@ OVER C!
          ELSE DUP RLNK@ FI
        ELSE DUP LLNK@ FI
        6 'D @ !
        OVER 0A + @ OVER RLNK! OVER 0A + !
        REPEAT
          POP POP OVER2 OVER SWAP -
            IF DUP C@ DUP
              IF OVER2 + 0FF AND
                IF OVER 3 + OVER + @ DUP C@
                  IF OVER2 0FF AND OVER C@ -
                    IF ROTCASE2
                    ELSE ROTCASE1 FI
                  ELSE ROTCASE3 FI
                  POP POP DUP PUSH SWAP DUP PUSH - 3 + !
                ELSE FAIL SWAP C! DROP FAIL FI
              ELSE DROP C! SUCCEED FI
            ELSE 2 + +! SUCCEED FI
          UNTIL
        ELSE DROP FI
      DROP
      ENDPROC
    ;
;S
```



FASL Credits

FASL arose in response to a need within FUNCTIONAL for a simple and efficient interpreter for system software development. An early FASL Manual (1977) was written with contributions from Eric Frey, Michel Julien, Roland Silver, and Ron Lebel. The idea of implementing the dictionary as a height balanced (AVL) tree came a year later, and with it the FASL TREE data type.

FASL was also made possible by the unselfishness of G. M. Adel'son-Vel'skiy and E. M. Landis, Donald E. Knuth, and Charles Moore.

The author has recently learned of two language processors which use AVL Trees for symbol tables, but not as a data type of the language: a MUMPS system (Dave Bridger for Tandem), and the IBM FORTRAN H Compiler. The current status of these language systems is not known by the author.

Special thanks to Kit Andrews for typing the manuscript on Functional's Wang Word Processor, and patiently illustrating the final versions of the Figures.

Assembler Listings for Search and Insertion

The following pages contain exerpts from the FASL listings pertaining to tree search and insertion for the 6800. Referring to these listings:

(1) The names used in the comments correspond to those used in Knuth's Algorithm 6.2.3A.

(2) The routines use variables HEAD and AVAIL to identify the tree and free nodes list on each invocation; the key should be in the eight byte area K.

(3) The variable VTV may be initialized to point to the default subroutine DEFNOT which causes a "failure" return on an insertion attempt to a full tree, or to a user supplied subroutine which allocates a new free nodes list (with at least one node) by placing the address of the list in AVAIL.

(4) Trees are initialized by placing a starting address in HEAD, an ending address in AVAIL, and calling the routine BTSIUP. On entry, AVAIL-HEAD should be greater than thirty-two, and zero mod sixteen. On exit, HEAD will not be modified and will point to the head node, and AVAIL will point to the free nodes list.

(5) All tree routines are object code relocatable.

(6) Quickie symbol table for these listings:

BTSIUP	E151	tree initialization
FINDIT	E168	tree search
BTSI	E17D	tree insertion
DEFNOT	E660	default tree overflow subroutine
K	D0	key for search & insertion, 8 bytes
HEAD	C2	pointer to tree
AVAIL	C4	pointer to free nodes list
VTV	C0	overflow transfer vector

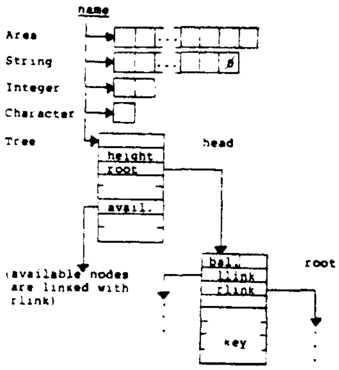
```

                          ; BALANCED TREE SEARCH AND INSERT

                          ; DIRECT MEMORY DATA DECLARATIONS

 00C0          VTV:       EQU 0C0          ; TREE OVERFLOW TRANSFER VECTOR TO SUBR ERROR HANDLI
 00C2          HEAD:      EQU VTV+2        ; POINTER TO TREE DESCRIPTER NODE
 00C4          AVAIL:     EQU HEAD+2       ; POINTER TO ROOT OF AVAILABLE NODES LIST
                                           ; ********************************************
                                           ; THE ABOVE THREE ITEMS ARE INPUTS TO BTSI
                                           ; VTV <- ADDRESS OF ERROR HANDLING SUBROUTINE
                                           ;        FOR OVERFOW OF ALLOTTED NODES
                                           ; HEAD <- POINTER TO TREE DESCRIPTER NODE, OR START
                                           ;         OF FREE SPACE FOR INITIALIZATION "BTSIUP"
                                           ; AVAIL <- POINTER TO LIST OF FREE NODES, OR END OF
                                           ;          FREE SPACE PLUS ONE FOR "BTSIUP"
                                           ; *****************
                                           ; BTSIUP USES HEAD AND AVAIL TO CREATE A NUL BALANC
                                           ; TREE AND A FREE NODES LIST.  AVAIL IS MODIFIED BY
                                           ; BTSIUP AND ALLOCT.
                                           ; ********************************************
 00C6          T:         EQU AVAIL+2
 00C8          S:         EQU T+2
 00CA          R:         EQU S+2
 00CC          Q:         EQU R+2
 00CE          P:         EQU Q+2
 00D0          K:         EQU P+2          ; KEY, EIGHT BYTES

                          ; NODE FORMAT
                          ;       NODE(0) BALANCE 1
                          ;       NODE(1) FLAG    1
                          ;       NODE(2) LEFTLINK 2
                          ;       NODE(4) RIGHTLINK 2
                          ;       NODE(6) VALUE   2
                          ;       NODE(8) KEY     8

                          ; FINDING AND INSERTING
 E11F 96 C8    NEXT:      LDAA S           ; S -> R<Q>
 E121 DE CC               LDX Q
 E123 A7 04               STAA X 04
 E125 E7 05               STAB X 05

 E127 DE C8               LDX S            ; INCREMENT S BY 16.
 E129 DF CC    INIT:      STX Q            ; Q IS PARENT OF S
 E12B D6 C9               LDAB S+1
 E12D CB 10               ADDB #10         ; THIS IS THE SIZE OF A NODE
 E12F 24 03               BCC SOK
 E131 7C 00C8             INC S             .
 E134 D7 C9    SOK:       STAB S+1         ; NOTE THAT B ALWAYS HAS S+1

 E136 DE C8               LDX S            ; CHECK AGAINST LIMIT
 E138 9C C4               CPX AVAIL
 E13A 26 E3               BNE NEXT

 E13C DE CC               LDX Q            ; SIGNAL END OF LIST
 E13E 6F 04               CLR X 04
 E140 6F 05               CLR X 05

 E142 96 C6               LDAA T           ; T -> R<HEAD> , SAVE POINTER TO ROOT
 E144 D6 C7               LDAB T+1
 E146 DE C2               LDX HEAD
 E148 A7 04               STAA X 04
 E14A E7 05               STAB X 05

 E14C DE CA               LDX R            ; R -> AVAIL , SAVE POINTER TO FREE LIST
 E14E DF C4               STX AVAIL
 E150 39                  RTS              ; END OF BTSI INITIALIZATION OF FREE LIST
                                           ; AND NULL TREE


                          ; BTSI INITIALIZATION

 E151 4F       BTSIUP:    CLRA
 E152 5F                  CLRB             ; MAKE A TREE DESCRIPTER, A NULL ROOT
 E153 DE C2               LDX HEAD         ; A FREE LIST BASED ON PARAMETERS
 E155 8D 5A               BSR SAXBI        ;      HEAD    START OF FREE SPACE
 E157 DF C6               STX T            ;      AVAIL   END OF FREE SPACE P
 E159 8D 56               BSR SAXBI        ; SAXBI CLEARS A NODE SINCE A=B=0
 E15B DF C8               STX S
 E15D DF CA               STX R            ; POINTER TO ROOT IN T
                                           ; POINTER TO AVAIL IN R
 E15F 20 C8               BRA INIT

 E161 EE 02    MOVLF:     LDX X 02         ; ALONG LEFTLINK
 E163 26 0D               BNE FIND

 E165 4F       NOTFND:    CLRA
 E166 4C                  INCA             ; FAILURE !! CC Z = 0
 E167 39                  RTS

                          ; LKUPPC -->

 E168 DE C2    FINDIT:    LDX HEAD
 E16A EE 04               LDX X 04         ; ROOT OF TREE
 E16C 20 04               BRA FIND

 E16E EE 04    MOVRF:     LDX X 04
 E170 27 F3               BEQ NOTFND       ; MOVE ALONG RIGHTLINK

 E172 DF CE    FIND:      STX P
 E174 8D 48               BSR KMP
 E176 22 F6               BHI MOVRF
 E178 26 E7               BNE MOVLF
 E17A 39                  RTS              ; SUCCESS !! CC Z = 1

 E17B 84 FF               ANDA #0FF        ; NOPE
```

```

                          ; BTSI BALANCED TREE SEARCH AND INSERT

 E17D DE C2    BTSI:      LDX HEAD         ; HEAD -> T
 E17F DF C6               STX T

 E181 EE 04               LDX X 04         ; R<HEAD> -> S , S POINTS TO REBALANCE
 E183 DF C8               STX S
 E185 20 21               BRA SEARCH

 E187 DE C0    OVRFLW:    LDX VTV          ; TREE OVERFLOW TRANSFER VECTOR
 E189 AD 00               JSR X 00

 E18B DE C4    ALLOCT:    LDX AVAIL        ; ALLOCATE A FREE NODE TO THE TREE
 E18D 27 F8               BEQ OVRFLW       ; CHECK FOR EMPTY FREE LIST

 E18F DF CC               STX Q            ; Q <- AVAIL

 E191 EE 04               LDX X 04         ; R<AVAIL> -> AVAIL
 E193 DF C4               STX AVAIL
 E195 DE CE               LDX P            ; SETUP PARAMETERS FOR CALLER
 E197 96 CC               LDAA Q
 E199 D6 CD               LDAB Q+1
 E19B 39                  RTS

 E19C A6 00    COMMOV:    LDAA X 00        ; CHECK BALANCE FACTOR
 E19E 27 06               BEQ FINMOV

 E1A0 DF C8               STX S            ; Q -> S
 E1A2 DE CE               LDX P            ; P -> T
 E1A4 DF C6               STX T
 E1A6 DE CC    FINMOV:    LDX Q            ; Q

 E1A8 DF CE    SEARCH:    STX P            ; -> P

 E1AA 8D 12               BSR KMP          ; K - K<P>
 E1AC 22 4D               BHI MOVR
 E1AE 26 3D               BNE MOVL
 E1B0 39                  RTS              ; SUCCESS!!!
                                           ; RETURN WITH CC Z = 1


 E1B1 8D 00    SAXBI:     BSR SAX4I
 E1B3 8D 00    SAX4I:     BSR SAX2I
 E1B5 8D 00    SAX2I:     BSR SAXINX
 E1B7 A7 00    SAXINX:    STAA X 00        ; STORE ACCUMULATORS INDEXED
 E1B9 E7 01               STAB X 01
 E1BB 08                  INX
 E1BC 08                  INX              ; POST INCREMENTED
 E1BD 39                  RTS


                          ; KEY COMPARE SUBROUTINE

 E1BE 96 D0    KMP:       LDAA K           ; K - KEY<X>
 E1C0 A1 08               CMPA X 08
 E1C2 26 28               BNE RTN          ; RETURN IF NOT EQUAL

 E1C4 96 D1               LDAA K+1
 E1C6 A1 09               CMPA X 09
 E1C8 26 22               BNE RTN

 E1CA 96 D2               LDAA K+2
 E1CC A1 0A               CMPA X 0A
 E1CE 26 1C               BNE RTN

 E1D0 96 D3               LDAA K+3
 E1D2 A1 0B               CMPA X 0B
 E1D4 26 16               BNE RTN

 E1D6 96 D4               LDAA K+4
 E1D8 A1 0C               CMPA X 0C
 E1DA 26 10               BNE RTN

 E1DC 96 D5               LDAA K+5
 E1DE A1 0D               CMPA X 0D
 E1E0 26 0A               BNE RTN

 E1E2 96 D6               LDAA K+6
 E1E4 A1 0E               CMPA X 0E
 E1E6 26 04               BNE RTN

 E1E8 96 D7               LDAA K+7
 E1EA A1 0F               CMPA X 0F
 E1EC 39       RTN:       RTS              ; DONE COMPARE OF EIGHT BYTES


 E1ED EE 02    MOVL:      LDX X 02         ; L<P> -> Q
 E1EF DF CC               STX Q
 E1F1 26 A9               BNE COMMOV       ; CONTINUE DOWN THE LEFT LINK

 E1F3 8D 96               BSR ALLOCT       ; DEAD END , ALLOCATE NEW NODE
 E1F5 A7 02               STAA X 02        ; L.LINK TO P
 E1F7 E7 03               STAB X 03        ; I.E. Q -> L<P>
 E1F9 20 0C               BRA INSERT

 E1FB EE 04    MOVR:      LDX X 04         ; R<P> -> Q
 E1FD DF CC               STX Q
 E1FF 26 9B               BNE COMMOV       ; CONTINUE DOWN THE RIGHT LINK

 E201 8D 88               BSR ALLOCT       ; DEAD END , ALLOCATE NEW NODE
 E203 A7 04               STAA X 04        ; R.LINK TO P
 E205 E7 05               STAB X 05        ; I.E. Q -> R<P>

 E207 DE CC    INSERT:    LDX Q            ; INITIALIZE THE NEW NODE
 E209 4F                  CLRA
 E20A 5F                  CLRB             ; CLEAR B<Q>, L<Q>, R<Q>, D<Q>, F<Q>
 E20B 8D A6               BSR SAX4I

 E20D 96 D0               LDAA K
 E20F D6 D1               LDAB K+1         ; K -> K<Q>
 E211 8D A4               BSR SAXINX

 E213 96 D2               LDAA K+2
 E215 D6 D3               LDAB K+3
 E217 8D 9E               BSR SAXINX

 E219 96 D4               LDAA K+4
 E21B D6 D5               LDAB K+5
 E21D 8D 98               BSR SAXINX

 E21F 96 D6               LDAA K+6
 E221 D6 D7               LDAB K+7
 E223 8D 92               BSR SAXINX
                                  ; ******************************
```

```

                                ; ADJUST BALANCE FACTORS ...
E225 DE C8     ADJ0:    LDX S           ; K - K<S>
E227 8D 95              BSR KMP
E229 22 06              BHI ADJ1

E22B C6 FF              LDAB #0FF       ; FLAG LT ( -1 -> A )
E22D EE 02              LDX X 02        ; L<S> ...
E22F 20 04              BRA ADJ2

E231 C6 01     ADJ1:    LDAB #01        ; FLAG GE ( 1 -> A )
E233 EE 04              LDX X 04        ; R<S> ...

E235 DF CA     ADJ2:    STX R           ; ... -> R
E237 20 10              BRA ADJ5        ; ENTER LOOP

E239 6F 00     ADJ3:    CLR X 00        ; 0 -> B<P>
E23B 8D 81              BSR KMP         ; K - K<P>             TIGHT!!!!!
E23D 22 06              BHI ADJ4

E23F 6A 00              DEC X 00        ; -1 -> B<P>
E241 EE 02              LDX X 02        ; L<P> ...
E243 20 04              BRA ADJ5

E245 6C 00     ADJ4:    INC X 00        ; 1 -> B<P>
E247 EE 04              LDX X 04        ; R<P> ...


E249 DF CE     ADJ5:    STX P           ; ... -> P
E24B 9C CC              CPX Q           ; UNTIL WE REACH Q
E24D 26 EA              BNE ADJ3
                                        ; *************


                                ; BALANCING ACT ...
E24F DE C8     BAL0:    LDX S           ; CHECK BALANCE FACTOR OF S
E251 A6 00              LDAA X 00
E253 26 07              BNE BAL1

E255 E7 00              STAB X 00       ; A -> B<S>

E257 DE C2              LDX HEAD        ; INCREMENT HEIGHT OF TREE
E259 6C 03              INC X 03
E25B 39                 RTS             ; FAIL!!!!
                                        ; RETURN CC Z = 0

E25C E1 00     BAL1:    CMPB X 00       ; CHECK B<S> AGAINST A
E25E 27 05              BEQ BAL2

E260 4F                 CLRA
E261 A7 00              STAA X 00       ; 0 -> B<S>
E263 4C                 INCA
E364 39                 RTS             ; FAIL!!!!
                                        ; RETURN CC Z = 0

E265 DE CA     BAL2:    LDX R           ; TREE NEEDS BALANCING
E267 5D                 TSTB
E268 2B 46              BMI BAL3

E26A E1 00              CMPB X 00       ; CHECK BALANCE FACTOR OF R
E26C 27 62              BEQ SROTL
                                        ; **********************

                                ; DOUBLE ROTATE LEFT ...
E26E EE 02     DROTL:   LDX X 02        ; L<R> -> P
E270 DF CE              STX P

E272 A6 04              LDAA X 04       ; R<P> -> L<R>
E274 E6 05              LDAB X 05
E276 DE CA              LDX R
E278 A7 02              STAA X 02
E27A E7 03              STAB X 03
E27C 6F 00              CLR X 00        ; 0 -> B<R>

E27E 96 CA              LDAA R          ; R -> R<P>
E280 D6 CB              LDAB R+1
E282 DE CE              LDX P
E284 A7 04              STAA X 04
E286 E7 05              STAB X 05

E288 A6 02              LDAA X 02       ; L<P> -> R<S>
E28A E6 03              LDAB X 03
E28C DE C8              LDX S
E28E A7 04              STAA X 04
E290 E7 05              STAB X 05
E292 6F 00              CLR X 00        ; 0 -> B<S>

E294 96 C8              LDAA S          ; S -> L<P>
E296 D6 C9              LDAB S+1
E298 DE CE              LDX P
E29A A7 02              STAA X 02
E29C E7 03              STAB X 03

E29E E6 00              LDAB X 00       ; CHECK BALANCE FACTOR OF P
E2A0 27 48              BEQ TUPLE
E2A2 2B 06              BMI SORL

E2A4 6F 00              CLR X 00        ; 0 -> B<P>
E2A6 DE C8              LDX S
E2A8 20 7E              BRA TUP0        ; -1 -> B<S>           V TIGHT !!!

E2AA 6F 00     SORL:    CLR X 00        ; 0 -> B<P>
E2AC DE CA              LDX R
E2AE 20 3C              BRA TUP1LE      ; 1 -> B<R>

                                ; ******
E2B0 E1 00     BAL3:    CMPB X 00
E2B2 26 3A              BNE DROTR       ; CHECK BALANCE FACTOR OF R


                                ; SINGLE ROTATE RIGHT ...
E2B4 DF CE     SROTR:   STX P           ; R -> P
E2B6 6F 00              CLR X 00        ; 0 -> B<R>
E2B8 A6 04              LDAA X 04
E2BA E6 05              LDAB X 05       ; R<R> -> L<S>
E2BC DE C8              LDX S
E2BE A7 02              STAA X 02
E2C0 E7 03              STAB X 03
E2C2 6F 00              CLR X 00        ; 0 -> B<S>

E2C4 96 C8              LDAA S          ; S -> R<R>
E2C6 D6 C9              LDAB S+1
E2C8 DE CA              LDX R
E2CA A7 04              STAA X 04
E2CC E7 05              STAB X 05
E2CE 20 62              BRA TUCHUP


                                ; SINGLE ROTATE LEFT ..
E2D0 DF CE     SROTL:   STX P           ; R -> P
E2D2 6F 00              CLR X 00        ; 0 -> B<R>
E2D4 A6 02              LDAA X 02
E2D6 E6 03              LDAB X 03       ; L<R> -> R<S>
E2D8 DE C8              LDX S
E2DA A7 04              STAA X 04
E2DC E7 05              STAB X 05
E2DE 6F 00              CLR X 00        ; 0 -> B<S>

E2E0 96 C8              LDAA S          ; S -> L<R>
E2E2 D6 C9              LDAB S+1
E2E4 DE CA              LDX R
E2E6 A7 02              STAA X 02
E2E8 E7 03              STAB X 03
E2EA 20 46     TUPLE:   BRA TUCHUP

E2EC 20 42     TUP1LE:  BRA TUP1

                                ; DOUBLE ROTATE RIGHT ...
E2EE EE 04     DROTR:   LDX X 04        ; R<R> -> P
E2F0 DF CE              STX P

E2F2 A6 02              LDAA X 02       ; L<P> -> R<R>
E2F4 E6 03              LDAB X 03
E2F6 DE CA              LDX R
E2F8 A7 04              STAA X 04
E2FA E7 05              STAB X 05
E2FC 6F 00              CLR X 00        ; 0 -> B<R>

E2FE 96 CA              LDAA R          ; R -> L<P>
E300 D6 CB              LDAB R+1
E302 DE CE              LDX P
E304 A7 02              STAA X 02
E306 E7 03              STAB X 03

E308 A6 04              LDAA X 04       ; R<P> -> L<S>
E30A E6 05              LDAB X 05
E30C DE C8              LDX S
E30E A7 02              STAA X 02
E310 E7 03              STAB X 03
E312 6F 00              CLR X 00        ; 0 -> B<S>

E314 96 C8              LDAA S          ; S -> R<P>
E316 D6 C9              LDAB S+1
E318 DE CE              LDX P
E31A A7 04              STAA X 04
E31C E7 05              STAB X 05

E31E E6 00              LDAB X 00       ; CHECK BALANCE FACTOR OF P
E320 27 10              BEQ TUCHUP
E322 2B 08              BMI DAUL

E324 6F 00              CLR X 00        ; 0 -> B<P>
E326 DE CA              LDX R           ; -1 -> B<R>
E328 6A 00     TUP0:    DEC X 00
E32A 20 06              BRA TUCHUP

E32C 6F 00     DAUL:    CLR X 00        ; 0 -> B<P>
E32E DE C8              LDX S           ; 1 -> B<S>
E330 6C 00     TUP1:    INC X 00

                                ; TOUCHUP ...
E332 96 CE     TUCHUP:  LDAA P          ; PREPARATION ...
E334 D6 CF              LDAB P+1

E336 DE C6              LDX T
E338 EE 04              LDX X 04        ; R<T> - S , COMPARE
E33A 9C C8              CPX S
E33C 27 09              BEQ TUP4

E33E DE C6              LDX T           ; P -> L<T>
E340 A7 02              STAA X 02
E342 E7 03              STAB X 03
E344 CA FF              ORAB #0FF
E346 39                 RTS             ; FAIL !!!!
                                        ; RETURN CC Z = 0

E347 DE C6     TUP4:    LDX T           ; P -> R<T>
E349 A7 04              STAA X 04
E34B E7 05              STAB X 05
E34D CA FF              ORAB #0FF
E34F 39                 RTS             ; FAIL !!!!
                                        ; RETURN CC Z = 0


                                ; ++++++++++++++++++++++++++++++++++++++++++++++++++++

E660                    ORG 0E660

E660 31        DEFNOT:  INS
E661 31                 INS
E662 31                 INS
E663 31                 INS
E664 4F                 CLRA            ; NO MORE ROOM IN TREE -> FAIL!
E665 39                 RTS

E666 01                 NOP
E667 01                 NOP
```

FASL HANDY REFERENCE

PREDEFINED DATA TYPES



size

zero to n bytes

one to n bytes

two bytes

one byte

thirty-two to n bytes

(each node is sixteen bytes)

(balance is one byte)
(each link is two bytes)
(key is eight bytes)
(three user defined bytes)

root (typical node)

Stack inputs and outputs are shown; top of stack on right.

Operand Key:	
n	two byte number
s	two byte signed number
u,addr, to, from	two byte unsigned number
b	one byte number or character
f	two byte boolean flag (zero or on
addr?	conditionally present addr
d	four byte signed number

(Digits are sometimes appended to these operand names.)
(Unless unsigned operands are indicated, arithmetic operations are twos complement.)

STACK MANIPULATION

DUP	(n -- n n)	Duplicate top of stack.
DROP	(n --)	Throw away top of stack.
SWAP	(n1 n2 -- n2 n1)	Reverse top two stack items.
OVER	(n1 n2 -- n1 n2 n1).	Make copy of second item on top.
OVER2	(n1 n2 n3 -- n1 n2 n3 n1)	Make copy of third item on top.
SROT	(n1 n2 n3 -- n2 n3 n1)	Rotate third item to top.
SWAPDROP	(n1 n2 -- n2)	Throw away second item on top.
DROP2	(n n --)	Throw away top two.
DROP3	(n n n --)	Throw away top three.
RPUSH	(n --)	Move top item to return stack.
RPOP	(-- n)	Retrieve top item from return stack.
'R	(s -- addr)	Compute address of sth byte on return stack.
'S	(s -- addr)	Compute address of sth byte on top (2 'S @ = OVER).

ARITHMETIC AND LOGICAL

+	(s1 s2 -- sum)	Add.
-	(s1 s2 -- difference)	Subtract (s1 - s2).
*	(s1 s2 -- product)	Multiply.
/	(s1 s2 -- quotient)	Divide (s1 ÷ s2).
MOD	(s1 s2 -- modulo)	Modulo (s1 mod s2).
MULE	(s1 s2 -- d)	Multiply extended.
DIVE	(d s -- quot mod)	Divide extended.
DIVMOD	(s1 s2 -- quot mod)	Divide modulus.
SEXT	(s -- d)	Sign Extend.
NEG	(s -- negation)	Negate.
ABS	(s -- absolute)	Absolute Value.
MIN	(s1 s2 -- min)	Minimum.
MAX	(s1 s2 -- max)	Maximum.
AND	(u1 u2 -- intersection)	Bitwise And.
OR	(u1 u2 -- conjunction)	Bitwise Or.
XOR	(u1 u2 -- disjunction)	Bitwise Exclusive Or.
NOT	(u -- complement)	Bitwise Inversion.
SUCCEED	(-- 1)	One (true).
FAIL	(-- 0)	Zero (false).
SHL	(n u -- n)	Shift Left (n, u times).
SHR	(n u -- n)	Shift Right (n, u times).
ROL	(n u -- n)	Rotate Left (n, u times).
ROR	(n u -- n)	Rotate Right (n, u times).

COMPARISON

		flag is true (one) if:
LTZ?	(s -- f)	Less than zero (s < 0)?
ZERO?	(n -- f)	Zero (n = 0)?
GTZ?	(s -- f)	Greater than zero (s > 0)?
LT?	(s1 s2 -- f)	Less than (s1 < s2)?
LE?	(s1 s2 -- f)	Less than or Equal (s1 ≤ s2)?
EQ?	(n1 n2 -- f)	Equal (n1 = n2)?
NE?	(n1 n2 -- f)	Not Equal (n1 ≠ n2)?
GE?	(s1 s2 -- f)	Greater than or Equal (s1 ≥ s2)?
GT?	(s1 s2 -- f)	Greater than (s1 > s2)?
ADDRGT?	(u1 u2 -- f)	Address Greater than? (u1 > u2)?
SLT?	(addr1 addr2 -- f)	String Less Than? (string at addr1 < string at addr2)?
SEQ?	(addr1 addr2 -- f)	Strings Equal? (string at addr1 = string at addr2)?

MEMORY

@	(addr -- n)	Replace address by contents.
!	(n addr --)	Store second item at address on top.
C@	(addr -- b)	Replace address by contents, one byte only (right justify zero padded).
C!	(b addr --)	Store right byte of second item at address on top.
+!	(n addr --)	Add second item to contents of address on top.
@SWAP	(addr1 addr2 --)	Swap contents of addr1 and addr2.
CMOVE	(from to u --)	Move u bytes in memory.
MOVE	(from to u --)	Move u double-bytes in memory.
SMOVE	(from to --)	Move string in memory.
LEAF	(addr1 addr2 -- addr? f)	Add key (string) at addr1 to tree at addr2. If f = true, then key was inserted at addr?, otherwise the key was already in tree (or tree is full).
FIND	(addr1 addr2 -- addr? f)	Locate key (string) at addr1 in tree at addr2. If f = true then key is at addr?, otherwise not found.
F$@	(addr -- b n)	Read data from tree node at addr.
F$!	(b n addr --)	Store data in tree node at addr.
'D	(s -- addr)	Compute address of nth byte in current Local Area.

CONTROL STRUCTURES

DO...LOOP	do: (end+1 start --)	Set up loop, give index range.
I	(-- index)	Place current index value on stack.
DO...+LOOP	+loop: (n --)	Like DO...LOOP except adds stack value (rather than one) to index.
IF...(true)...FI	if: (f --)	If top of stack true (non-zero), execute.
IF...(true)...ELSE...(false)...FI		Same, but if false, execute ELSE clause.
DO...IF...(true)...LOOP ELSE...(false)...EXIT FI		The EXIT in ELSE clause terminates loop prematurely. +LOOP may be used in place of LOOP, and the LOOP and EXIT words may be reversed.
REPEAT...UNTIL	until: (f --)	Loop back to REPEAT until true at UNTIL.
WHILE... CONTINUE...(true)... WHILEND... (false)...	continue: (f --)	Continue while true at CONTINUE, otherwise leave loop; WHILEND loops unconditionally.

INPUT/OUTPUT

MESS	(addr --)	Type message (string) at addr.
TYPE	(addr b --)	Type message at addr terminated by byte b.
=	(n --)	Type number on top of stack.
C=	(b --)	Type one byte number on top.
CRLF	(--)	Type a Carriage Return, Line Feed.
SP	(--)	Type a Space.
DUMP	(addr u --)	Type u bytes starting at addr.
PRTREE	(addr --)	Type tree at addr.
GETKEY	(--)	Read characters until delimiter to Global Area K.
CHECKKEY	(-- f)	True if K is non-numeric.
CONVERTK	(-- n)	Converts string at K to number.
ASK	(addr delim count --)	Read characters to addr until delimiter or count.
WORD	(addr delim --)	Read characters to addr until delimiter.

DEFINING WORDS

: xxx	(--)	Begin colon-word definition of xxx.
;	(--)	End colon-word definition.
, xxx	(addr --)	Used to name machine language operation.
GLOBAL xxx	(n --) xxx: (-- addr)	Create Global Variable xxx with initial value n; returns address when executed.
CONSTANT xxx	(n --) xxx: (-- n)	Create Constant Variable xxx with value n; returns value when executed.
AREA xxx	(n --) xxx: (-- addr)	Create Global Area xxx of size n, with no initial value; returns address when executed.
" xxx ...	(--) xxx: (-- addr)	Create Global String xxx with initial value of text typed in after xxx delimited by quote ("); returns address when executed.
TREE xxx	(n --) xxx: (-- addr)	Create Global Tree xxx of size n nodes, and initialize; returns address when executed.
TREEINIT	(addr1 addr2 --)	Initialize Tree from addr1 to addr2-1 (used for Local or preallocated Trees).
PROC...ENDPROC	proc: (n --)	Allocate/Deallocate n bytes of Local Area on return stack (*only* used inside colon-words).

SYSTEM & MISCELLANEOUS

LOAD...;S	load: (addr1 addr2 --)	LOAD modifies current Input pointers (addr 1 is address of input string, addr2 is address of machine level input *subroutine*), *;S restores* previous values (uses return stack...be careful).
(	(--)	Begin Comment, delimited by right paren. (up to 8K characters are allowed).
PGMOVE	(u1 u1 --)	Block Move of 8Kbytes from page u1 to page u2.
INTO	(u --)	Block Move from Inbox to page u.
OUTOF	(u --)	Block Move from Page u to Outbox.
PEREAD	(u --)	DUMBOS Read from Outbox of Cyblok u to Inbox.
DUMBWAIT	(--)	Wait for DUMBOS command slot acknowledge.
PEMESS	(addr u --)	*Send* message at addr to Cyblok u.
PEWAIT	(addr u --)	Receive message from Cyblok u to addr.
PESLOT	(u -- addr)	Compute inslot address for Cyblok u.

LETTERS

I would like to point out a possible misconception that I noticed in one of the judge's comments on page 54 in the special FD on Case Structures. The third itemlisted as an "advantage" states "(The) case selector is kept on (the) return stack instead of in a special variable. This allows nesting of CASE constructs." I'd like to point out that the FORTH-85 CASE structure, which uses a variable (VCASE), is also nestable. The reason for this is that once a match has been made and execution is in progress between, CASE . . .END-CASE the contents of VCASE have served their purpose. Further nesting at this point can alter the contents of VCASE without problems. When the unnesting occurs, END-CASE shoots the Forth instruction pointer to the words after the end of the case structure. END-CASE does not need the older contents of VCASE. If the programmer would like to retain the selector value, a simple "VCASE @" directly after CASE will preserve the contents of the stack. Then, for any following Forth words having nested DO-CASE structures, the problem of overwriting is solved. The variable storage method takes a little longer to retrieve the current selector value (i.e. VCASE @ versus DUP, or versus I), but retrieving VCASE has not been very common in my experience. To me VCASE @ is more self-explanatory in the context of the program than either DUP or I. In addition, my feeling is that messing up the return stack so the normal index values (I & J) cannot be used within a CASE. . . END-CASE phrase, is a definite disadvantage. To solve return stack problems like this, advanced Forth Systems, such as the one now at Kitt Peak or STOIC, have three stacks. The extra stack is used explicitly for LOOP indices while the rturn stack is used for return addresses and temporary storage. In lieu of a third stack, the VCASE variable presents a clear way of handling this situation. The variable storage method would need to be changed to user variable storage if multi-tasking was to be implemented. This is only slightly more complicated than the current version. In my extension, I tried both return stack and variable methods. I selected the variable storage due to speed improvements as well as the aguments above. Also, in regards to speed, the CALL's and JMP's within the code statement for CASES are weak in style snce the objective in code statements is speed. These really should be expanded out (i.e. MACRO'd!). My original intent was to make the article do double duty be demonstrating these techniques as a stepping stone to some debugging methods I came up with.

Bob Giles
Tulsa, OK

THE EXECUTION VARIABLE AND ARRAY:

Michael A. McCourt
University of Rochester

A useful programming construct is the jump table or 'COMPUTED GO TO' type of structure. In Forth the execution variable and array can be used. The Forth word EXECUTE executes the code address on the top of the stack. If one defines:

```
: XEQ <BUILDS , DOES> @ EXECUTE;
```

a word containing a code address as its parameter can be created. As an example

```
: TEST ." THIS IS A TEST" CR ;
0 XEQ FRED   ' TEST CFA ' FRED 2+ !
```

The word TEST can now be executed by typing FRED. You might ask--why not type TEST to execute TEST? The reason is that FRED is now a variable--of sorts. By changing the contents of the parameter stored in FRED the action of FRED can be changed. Execution arrays are similar, however, here several code addresses can be stored and later accessed by index number. In our Forth system (an updated URTH system to Forth-79 running on a PDP-11) the Forth code address of zero is disallowed and will cause execution of the current ABORT procedure which itself is contained in a variable, i.e.

```
: ABORT ABEND @ EXECUTE ;
```

All execution variables and arrays are initialized to zero so that they will have predictable results.

Three words shown in block 502 listed below are used to change the contents of execution variables and arrays.

INSTALL <name>

returns the code field address of <name>.

<code addr> IN <XEQ var name>

stores the code address in the parameter field of XEQ name.

<code addr><array offset> OFFSET.IN
< ()XEQ array name>

stores the code address at the offset in the ()XEQ array.

Thus the previous example could be written as

```
0 XEQ FRED   INSTALL TEST IN FRED
```

Note that INSTALL and IN work within a colon definition, e.g.,

```
: DUMMY ;
: TURN.ON INSTALL TEST IN FRED;
: TURN.OFF INSTALL DUMMY IN FRED;
```

Execution variables are useful for a variety of functions such as creating forward references, switching output and/or input routines among several terminals, debug routines and of course implementing a jump table.

Examples

1. JUMP TABLE

Problem:

Define a function that will perform one of 26 operations depending on which control key was typed.

Possible Solution:

```
26 ()XEQ CTRL.KEY
```

```
INSTALL 1FUNCTION 1 OFFSET.IN CTRL.KEY
INSTALL 2FUNCTION 2 OFFSET.IN CTRL.KEY
        .
        .
        .
INSTALL 26 FUNCTION 26 OFFSET.IN
   CTRL.KEY

: OPERATOR? BEGIN KEY DUP 27 <=
   IF CTRL.KEY ELSE DROP THEN AGAIN;
```

One could implement the above with a case or select statement, but the execution array has less overhead in execution speed and memory usage.

2. MULTITERMINAL DRIVERS

Problem:

One has a video terminal with addressable cursor and a 'dumb' hardcopy terminal. The latter terminal does not accept cursor control characters gracefully.

Possible Solution:

One solution which alleviates this problem is shown listed below in block 500. (Publ. note: we're not printing block 500.) The word CTRL is an execution variable. When the video terminal is operating (TT1) all control characters are EMIT'ed; however, when the printer is installed (TT0) the control characters are DROP'ed.

The words EMIT and KEY are defined as state variables as is ABEND (user variables might be a familiar name to some) and are addressed for multitasking. They permit each task access to its own terminal driver.

```
: TEST2 0 0 TPC ." TESTING" ;
   ( POSITION CURSOR AND PRINT )


TT1 TEST2  ( 'TESTING' WILL START AT
   POSITION <0,0> )

TT0 TEST2  ( CONTROL CHARACTERS FOR
   0 0 TPC HAVE NO EFFECT)

   22 LIST ( LISTING SENT TO PRINTER )
TT1        ( BACK TO DISPLAY )
```

3. FORWARD REFERENCE

At times early in an application program one needs to define an error handling routine. However, since none of the higher level words have been defined the error handling is rather primitive. Execution variables allow one to 'leave a blank' for the error routine.

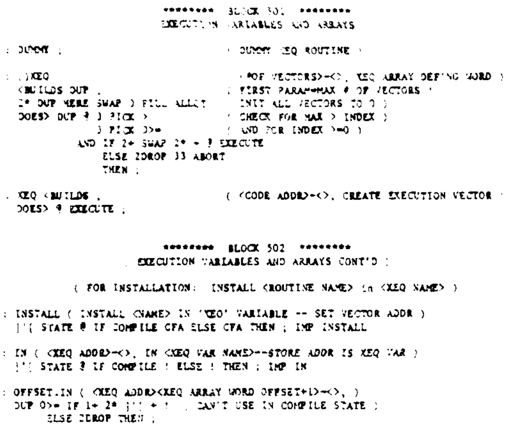
Suppose one has

```
     0 XEQ DERROR

<device function code>
 : DIO GO.BIT OR DEVICE.CONTROL !
   WAIT.FOR.DEVICE.DONE
   DEVICE.STATUS @ 0< IF DERROR THEN ;
```

Assume DIO is for control of a mag tape drive. At this point in the application program DERROR would normally be able to do only an ABORT. With a tape drive one would prefer to have some sort of recovery procedure on write errors to either delete the last file or at least write an End of File mark. With the execution variable one can install such a high level routine at a later time after all the necessary words (such as skip record, read record, and write EOF) have been defined. DERROR could also be defined as an ()XEQ array and each error would have its own associated error handling.

The previous examples demonstrate the power of the <BUILDS ... DOES> Forth constructs. XEQ and ()XEQ are just two examples of defining words. It is possible to build a wide range of such defining words from words that build simple linear arrays to ones that define complex relational data bases. In all cases one is associating a data structure (here, a simple code address) with an algorithm for using the data (here, EXECUTE the code address) and as Wirth has written DATA STRUCTURES + ALGORITHMS = PROGRAMS*

*Wirth, Niklaus, "Algorithms + Data Structures = Programs," Englewood Cliffs, Prentice-Hall, Inc. 1976.



```
                 ********  BLOCK 501  ********
                 EXECUTION VARIABLES AND ARRAYS

: DUMMY ;                         ( DUMMY XEQ ROUTINE )

: ()XEQ                            ( #OF VECTORS>-<>, XEQ ARRAY DEF'NG WORD )
  <BUILDS DUP ,                   ( FIRST PARAM=MAX # OF VECTORS )
  2* DUP HERE SWAP 0 FILL ALLOT   ( INIT ALL VECTORS TO 0 )
  DOES> DUP @ 3 PICK >            ( CHECK FOR MAX > INDEX )
            3 PICK 0>=            ( AND FOR INDEX >=0 )
        AND IF 2+ SWAP 2* + @ EXECUTE
            ELSE 2DROP 33 ABORT
            THEN ;

: XEQ <BUILDS ,                   ( <CODE ADDR>-<>, CREATE EXECUTION VECTOR )
  DOES> @ EXECUTE ;

                 ********  BLOCK 502  ********
              ( EXECUTION VARIABLES AND ARRAYS CONT'D )

        ( FOR INSTALLATION:  INSTALL <ROUTINE NAME> in <XEQ NAME> )

: INSTALL ( INSTALL <NAME> IN 'XEQ' VARIABLE -- SET VECTOR ADDR )
  ]'[ STATE @ IF COMPILE CFA ELSE CFA THEN ; IMP INSTALL

: IN ( <XEQ ADDR>-<>, IN <XEQ VAR NAME>--STORE ADDR IS XEQ VAR )
  ]'[ STATE @ IF COMPILE ! ELSE ! THEN ; IMP IN

: OFFSET.IN ( <XEQ ADDR><XEQ ARRAY WORD OFFSET+1>-<>, )
  DUP 0>= IF 1+ 2* ]'[ + !  ( CAN'T USE IN COMPILE STATE )
      ELSE 2DROP THEN ;
```

MEETINGS

NORTHERN CALIFORNIA

8/23/80

Ray Dessey, a chemist from Virginia Polytechnical Institute in Blacksberg, was visiting and he described his recent trip to China. FORTH accompanied him embodied in an AIM and students at Futan University, Shanghai, got a taste of FORTH. Dr. Dessey said the University already had 3 LSI-11's with Pertec floppies. He also described Virginia Tech's teaching/research machine which is a network with 3 three terminal hosts each having 15 satellite processors. FORTH runs under an RT-11 operating system. Instrumentation simulation (a function generator + noise) is one use.

Bill Ragsdale announced the Asilomar FORTH retreat (cf., FD Vol. II No. 3 for details).

Kim Harris described OPTIMIST, a program which reminded me of a cantankerous ELIZA. This FORTH program, originally written in PL/1 by Kildall, exemplifies a SECURED vocabulary as part of Kim's tutorial on PRIVATE VOCABULARIES. He showed how they are produced, tested and sealed.

Howard Pearlmutter discussed FIGGRAPH and the "human interface" of FORTH. The FIGGRAPH committee is to generate and articulate hardware specs, goals, and a vocabulary. Howard advised us to attend the HOME BREW COMPUTER CLUB's showing, via a G.E. LIGHT VALVE, of computer graphics. (I saw it and it was as entertaining as LASERIUM).

Handouts included:

- Harris' OPTIMIST and PRIVATE VOCABULARY support
- Zimmer's TERMINAL, a program to teach a FORTHed Ohio Scientific Instruments OS-650v3 to act dumb
- FORTH MODIFICATION LABORATORY's CALL FOR PAPERS: (Programming methodology, Virtual Machine Implementation, Concurrency, Language & Compiler, Applications, and Standardization.

HELP WANTED

SENIOR PROGRAMMER to produce new polyFORTH systems and applications.

Contact: Carol Ritscher
FORTH, Inc.
2309 Pacific Coast Hwy.
Hermosa Beach, CA 90254

PROJECT BENCHMARK

A small, informal group of microcomputer enthusiasts here in Albuquerque read with interest "Project Benchmark" in the June issue of the magazine "INTERFACE AGE." We have amongst us a variety of systems and languages, including 8080, 6800, and the AM-100, interpreter and compiler versions of BASIC, and fig-FORTH on the three system types. We ran the benchmark program all around and have attached the results of our testing.

We found the results to be most interesting and offer them to the members of the Forth Interest Group. In addition to the timing results, there was also a significant advantage in memory for the FORTH programs. The compiled AlphaBasic program size was 192 bytes while the FORTH benchmark program size was 166 bytes. All three implementations of FORTH were based on the fig model, and the program ran without modification on all systems demonstrating the transportability achievable with FORTH.

I have attached a listing of the FORTH program. The implementation of the language for the 8080 and the 6800 were from fig, while the Alpha Micro version was provided by Sierra Computer Co., Albuquerque, NM.

George O. Young III
Albuquerque, NM

```
RUN.BENCH   OK
1000 RUN.BENCH
STARTING
   1   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67
  71  73  79  83  89  97 101 103 107 109 113 127 131 137 139 149 151 157 163 167
 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277
 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523
 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653
 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797
 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937
 941 947 953 967 971 977 983 991 997
FINISHED
```

INTERFACE AGE Benchmark Program
Results from the Albuquerque Group

CPU/System	Clock	Language	Execution Time
6800	.9 mhz	FORTH	6' 13"
8080 North Star DOS	1.84 mhz	FORTH	5' 49"
AM-100	2 mhz	FORTH	2' 53"
AM-100 w/ polled serial I/O	2 mhz	AlphaBasic	9' 37"
8080 Heath H8	2.2 mhz	Benton Harbor Basic	42'
8080 North Star DOS	1.84 mhz	MicroSoft Basic	21' 8"
8080 North Star DOS	1.84 mhz	MicroSoft Compiler Basic	8' 41"
8080 North Star DOS	1.84 mhz	North Star Basic	41' 13"
8080 North Star DOS	1.84 mhz	C-Basic V1.01	77'
Z-80 SuperBrain	?	C-Basic	53'
6502 Ohio Scientific	2 mhz	MicroSoft Basic	8' 25"
Z-80 North Star	4 mhz	North Star Basic	19'
Z-80 North Star w/ Floating Point Board	4 mhz	North Star	11' 25"
6800	.9 mhz	PERCOM Super Basic	73'
6800	.9 mhz	SWTP V2.3 8k Basic	81'
CYBER 176	?	FORTRAN	190 ms
CYBER 176	?	PASCAL	260 ms
CDC 6600	?	FORTRAN	1069 ms
CDC 6600	?	PASCAL	3500 ms

NOTE: Although speed improvements may be made to the basic algorithm as published in INTERFACE AGE, the programs used in the above test remained a true representation of the algorithm published in the June issue of INTERFACE AGE magazine.

```
              **************  FIG-FORTH  ************
             **** FOR THE ALPHA MICRO SYSTEM ****
                        fig-FORTH V 1.1
                     AM-FORTH VERSION A.4
               Property of SIERRA COMPUTER COMPANY
                          617 Mark NE
                     Albuquerque, NM  87123
                          12 July 1980

SCR # 8
 ( INTERFACE AGE BENCHMARK PROGRAM ENTER W/ UPPER LIMIT-1 * )
 : BENCH DUP 2 / 1+ SWAP ." STARTING " CR
         1 DO DUP I 1 ROT
           2 DO DROP DUP I /MOD
                DUP 0= IF DROP DROP 1 LEAVE
                       ELSE 1 = IF DROP 1
                                ELSE DUP 0 > IF DROP 1
                                             ELSE 0= IF 0 LEAVE
                                                     ENDIF
                                             ENDIF
                                ENDIF
                       ENDIF
                LOOP
           IF 4 .R ELSE DROP ENDIF
           LOOP DROP CR ." FINISHED. "  ;
              -->

SCR # 9
 ( INTERFACE AGE BENCHMARK PROGRAM, CALLING ROUTINE         * )
 : RUN.BENCH  TIME SWAP BENCH TIME
              SWAP - 60 / 60 /MOD
              CR ." ELAPSED TIME: "
                . ." MINUTES,  "
                . ." SECONDS.  "    ;
```

HELP WANTED

FORTH PROGRAMMERS (or ASSEMBLY programmers who want to learn FORTH).
Contact: Gary Osumi (714) 453-2345
Hydro Products, San Diego, CA

IPS
A GERMAN FORTH-DIALECT

Dr. Karl Meinzer
Marbach, W. Germany

The AMSAT-Phase III communication satellites for radio-amateurs utilize a computer on board for a variety of tasks. In order to simplify the programming and to allow a simple dialogue with the spacecraft the language IPS was developed (in 1976). It is a Forth-derivative geared very strongly towards engineering applications (real-time control) and by now it is also used in a variety of control-related areas. The following lines describe the rationale of the system and its main differences as compared to FORTH.

Area of Application

The IPS development was aimed in particular towards the "low" end of computers. Most control applications do not justify a larger computer for cost reasons. On the other hand, these applications profit most from a powerful language processor since the common techniques are very clumsy to use. The computer I had in mind when I designed IPS was at about the level of the TRS-80 with 16K bytes of RAM (integral video memory and cassette for mass storage). For real-world interactions control-I/O and a 20ms interrupt must be added to complete the system.

The IPS Language

An introduction to IPS was given in BYTE, Jan. 1979, pp. 146; so here I want to explain the difference to FORTH. First: for the names I tried to find words which are more logical in a postfix environment. Take the IF ELSE THEN construct, e.g., in IPS it is replaced by YES? NO: and THEN. This seemed more logical since the IF implies a test following. But with the preceding test YES? is more appropriate. Of course these fine points may not be very important. Others are more so: numbers used an truth-variable on the stack use only the least significant bit. This allows the 16-bit logic operators like AND OR or XOR to be used consistently with truth-variables.

A major difference is the way names are encoded. I did not like the limitations coming from the 3 characters plus length codes; but then neither did I want to use more than 4 bytes for the code. The following technique was adopted: from all characters of the name (up to 63), a division remainder using the polynomial X24 + X7 + X2 + X1 + 1 is computed (3 bytes) and stored with the length of the name. This technique allows abitrary names; e.g., MACHINE-A1 and MACHINE-A2 are distinct and not confused by the system.

Theoretically there is a small (10 to the -7) probability of a collision --in practice I never yet encountered one. In any case, no harm can come from this because in IPS the system does not allow the redefinition of names. This "advantage" of FORTH was dropped very early because from our user-feedback it soon became clear that it was--directly or indirectly-- one of the major causes for programming errors.

Other plausibility checks were added to make the system more forgiving against the typical programming blunders. (I do not believe in the FORTH-assumption that the programmer can be perfect--I am a good example to the contrary). In fact, a few checks can make the system virtually crashproof. Of course, one has to be careful not to get carried away with this--if the integrity of the system is reduced, much of the power of a FORTH-like language goes away.

Three examples within IPS:

- During definitions the colon puts an unused address on the stack. The semicolon checks for this number: if it finds a different number, most likely a structuring error has occurred. The definition is removed and an error message is written.

- Each word has a unique 2-bit identification in the name field defining its use in the interpretive mode. Words like YES?, for example, are not executed outside definitions--so no "magic effects" can result.

- The number of interpreter states the programmer has to keep in mind is minimized. The base for number conversions is set explicitly. Numbers like 40 or -721 are treated as decimal, #03 or #AF07 as hexadecimal numbers.

Real-Time Multiprogramming

The typical situation with real-time control has the processor waiting for some event, then executing a task --usually very fast--and then again waiting for other events. In practice, typically the computer must attend to a number of such tasks. This allows for a fairly simple multiprogramming concept. The tasks are put in a cyclic "chain," an array containing the addresses of the tasks to be executed. The system executed them periodically in a roundrobin fashion. Provided that none of the tasks "grabs" the processor this results in a reasonably fair arbitration of processor time and was found sufficient for most control applications. Two operators are provided to allow dynamic and static task allocations: INCHAIN and DECHAIN.

The interpreter/compiler is also a task in this sense--it executes one word at a time before it returns to the chain. This keeps all the debugging capability of the interpreter a hand while other tasks are executing.

The system is augmented by the concept of "pseudo-interrupts." The address interpreter (NEXT) is effectively a stack-machine which has ideal properties for interrupting it--no saving is required. If the address interpreter can accept these pseudo-interrupts between the execution of code-routines, a very powerful high-level interrupt-concept is possible. In IPS such a pseudo-interrupt is executed every 20ms to keep the keyboard alive and for timekeeping purposes. Other pseudo-interrupts may be added as required.

Signalling to the address interpreter the pseudo-interrupt request without creating additional overhead is a bit involved with most processors. Only with the CDP 1802, this is straightforward--the address interpreter contains a jump that can be made conditional on an external signal (External flag). With the other processors a real interrupt is used to modify the code of NEXT; admittedly a less than desirable way of programming. Since this occurs only at a single point, it was considered to be the lesser evil over a possibly increased duration of NEXT.

Handling and Testing

IPS is strongly TV-screen oriented. This allowed the stack to be continuously visible by putting a display-program into the chain. For debugging it is a great help not having to request the stack-content, but seeing it continuously. During the operation of chain-operators the system remains "live," you always can go after problems and investigate.

Typically, programs are first written on cassette with the integral text-editor as blocks of 512 bytes each. Then the blocks are compiled and tested. If necessary, blocks may be edited on the cassette and recompiled to solve bugs. Eventually a binary dump of the whole program (IPS plus application) is produced to facilitate fast reloading.

Experiences So Far

Primarily, the system was developed for the Phase III spacecraft that was launched in May 1980. It gave the handling of the satellite an unprecedented degree of flexibility and at the same time helped to solve the rather complex attitude control problems with a minimum of pain. The spherical trigonometry of the satellite was solved very elegantly by Cordic-type rotation operators rather than the conventional solution using sines and cosines. This allows a geometrical analysis of the problems rather than the much more complicated alebraic analysis.

Unfortunately the launcher (ARIANE L02) failed and the spacecraft was destroyed--a repeat is scheduled for early 1982. The ground equipment also uses IPS. An English version for the 8080 using an S-100 bus computer was used for the safety surveillance computer.

Furthermore, a large number of COSMAC based computers within the University of Marburg utilize IPS for a number of research-data-acquisition tasks. All in all, our experience with the system has fully met our goals--to simplify real-time control.

The Problem of Distribution

With the real-time capabilities of IPS, portability of the system is much more difficult to achieve than with more common language processors--the hardware configurations have much more connections with the system than say with a BASIC interpreter. Typically we modify the IPS meta-source to match the hardware at hand and then run the source through a meta-compiler producing the new system. The lack of suitable "standard-computers" having the required real-time hardware extensions so far has prevented a very widespread distribution of IPS. Now we have a version running on the TRS-80 with a few restrictions; by adding some hardware these restrictions go away. As a next step we intend to build a meta-compiler running on an unmodified TRS-80. Hopefully this way we can get "out of the cycle" and thus enable a widespread distribution of IPS. The large number of letters I received after the BYTE paper convinced me that the need for such a system is very real. I should be pleased if this letter also presents a stimulus to FORTH programmers to add some of the IPS concepts to enhance its usefulness for real-time control.

AUTHORS WANTED

Mountain View Press, the source for printed FORTH, will publish, advertise and distribute your FORTH in printed form. Substantial royalty arrangement.

Contact: Roy Martens
Mountain View Press
PO Box 4656
Mt. View, CA 94040

HELP WANTED

PROJECT MANAGER to supervise applications and special systems projects.
Contact: Carol Ritscher
FORTH, Inc.
2309 Pacific Coast Hwy.
Hermosa Beach, CA 90254

THE CASE, SEL, AND COND STRUCTURES:

Peter H. Helmers
University of Rochester

The following is a description of the three "case-like" structures which have been added to URTH for the Ultrasound Lab in the Department of Radiology at the University of Rochester. These three structures were evolved from a simpler prototype CASE statement developed by Rich Marisa at the University's Towne House Computer Center and by Larry Forsley at the University's Laboratory for Laser Energetics.

Execution Time Operation

The three structures to be described are the CASE, SEL and COND statements. Referring to the examples given in figure 1, it can be seen that each of these structure types consists of a series of one or more clauses delimited by the << and >> words, and enclosed within the appropriate structure defining words:

```
    CASE ... ENDCASE
    SEL  ... ENDSEL
or, COND ... ENDCOND
```

Each can have an optional OTHERWISE clause which is executed if none of the other clauses is executed.

These structure types differ in how a given clause is selected for execution; thus the description of each type which follows will try to elucidate their difference.

The COND structure is a more readable syntax for a series of nested IF...ELSE...THEN statements. The COND structure consists of a series of clauses with explicitly specified conditions and associated actions which are executed if the condition is satisfied. Only the first clause whose condition is met is executed in a given execution of the structure. The integer on the top of the parameter stack is destroyed after execution. The TEST-COND definition shown in figure 1 is an example of the syntax of this structure.

The SEL structure is similar to the COND structure except that it uses an implicit test for equality to an explicitly specified integer value. Thus when the top of the parameter stack value matches that used within the SEL clause, the associated action is taken. As with the COND statement, only the first clause selected will be executed in a single pass through the structure. Additionally, the integer value tested is removed from the top of the stack after execution. An example of this structure is the TEST-SEL definition shown in figure 1.

The CASE structure is in turn similar to the SEL structure except that it uses both an impliclit test for equality, and an implicit numbering of the case clauses, starting with 1 for the first clause. Thus an explicit test value does not have to be specified. In operation, for example, a value of three on the top of the parameter stack would cause execution of the third clause in a CASE statement, if it exists. Note that the CASE value on the top of the parameter stack is dropped after each pass through the structure.

Compiler Operation

The words <<, WHEN, and >> are used in common by all three types of structures; thus these words' compiling operations are dependent on the type of structure being used. This "type" information is determined by the integer on the top of the parameter stack at compile time--which is set in turn by the words: CASE, SEL, or COND. These structure defining words each put two integer values on the stack. The next to top of the stack value is a flag value of zero which is used by the structure terminating words (ENDSEL, etc.) when they link up branch addresses. The top of stack value reflects the type of structure being used as summarized here:

-2	COND structure
-1	SEL structure
≥0	CASE structure; this integer is actually the value of the previous CASE clause which was compiled.

The <<, WHEN, and >> words thus analyze the top of stack value to determine what words are to be compiled into the new word's parameter list. For example, WHEN for a SEL structure compiles the words OVER = and IF into the new word's definition.

The examples of the structures in figure 1 illustrate their respective syntaxes. Figures 2 through 4 are outputs from a FORTH debugger (decompiler) which emphasize the different compilations of <<, WHEN, and >> for each type of structure. (Note that the results of the compilation process are listed to the left, while the corresponding high level compiler words are at the right.) By studying the definitions of these structural words in figure 5 in conjunction with the examples and the debugger outputs, operation should be easily adapted to other FORTH systems.

```
OK DEBUG TEST-COND
TEST-COND LINKED TO 332D
: DEFINITION
3376 1439 DUP -----------------   <<
3378 0111 LIT  FFFE
337C 17DB <
337E 07FD $IF  3388 -----------   WHEN
3382 32B7 LESS-THAN-NEG-TWO
3384 0810 $ELSE  339A ---------   >>
3388 1439 DUP -----------------   <<
338A 1361 2
338C 1806 >=
338E 07FD $IF  3398 -----------   WHEN
3392 32CF GREATER-THAN-ONE
3394 0810 $ELSE  339A ---------   >>
339B 1A6B CR
339A 13BB DROP ----------------   ENDCOND
339C 01C8 $;
OK
```

FIGURE 2

```
( STRUCTURE EXAMPLES - PHH - 8 22 80 )
: FIRST ;
: SECOND ;
: THIRD ;
: WHO-KNOWS? ;
: ONE ;
: NEG-THIRTY-THREE ;
: FIVE ;
: LESS-THAN-NEG-TWO ;
: GREATER-THAN-ONE ;

( STRUCTURE TESTS - CON'T - PHH - 8 22 80 )
: TEST-CASE
   CASE
       << FIRST >>
       << SECOND >>
       << THIRD >>
   OTHERWISE WHO-KNOWS?
   ENDCASE ;

: TEST-SEL
   SEL
       << 1 WHEN ONE >>
       << -33 WHEN NEG-THIRTY-THREE >>
       << 5 WHEN FIVE >>
   OTHERWISE WHO-KNOWS?
   ENDSEL ;

: TEST-COND
   COND
       << -2 < WHEN LESS-THAN-NEG-TWO >>
       << 2 >= WHEN GREATER-THAN-ONE >>
   OTHERWISE CR
   ENDCOND
;
```

FIGURE 1

```
OK DEBUG TEST-SEL
TEST-SEL  LINKED TO 32E3
: DEFINITION
332D 07B4 1
332F 142C OVER          )
3331 17BE =             )------   WHEN
3333 07FD $IF  333D     )
3337 327A ONE
3339 0810 $ELSE  3363 ---------   >>
333D 0111 LIT  FFDF
3341 142C OVER          )
3343 17BE =             )------   WHEN
3345 07FD $IF  334F     )
3349 3292 NEG-THIRTY-THREE
334B 0810 $ELSE  3363 ---------   >>
334F 0111 LIT  0005
3353 142C OVER          )
3355 17BE =             )------   WHEN
3357 07FD $IF  3361     )
335B 392E FIVE
335D 0810 $ELSE  3363 ---------   >>
3361 326F WHO-KNOWS?
3363 13BB DROP ----------------   ENDSEL
3365 01C8 $;
OK
```

FIGURE 3

```
OK DEBUG TEST-CASE
TEST-CASE  LINKED TO 32D2
: DEFINITION
32E3 0111 LIT  0001      )
32E7 142C OVER           )
32E9 17BE =              )------   <<
32EB 07FD $IF  32F5      )
32EF 3242 FIRST
32F1 0810 $ELSE  331B ---------   >>
32F5 0111 LIT  0002      )
32F9 142C OVER           )
32FB 17BE =              )------   <<
32FD 07FD $IF  3307      )
3301 3250 SECOND
3303 0810 $ELSE  331B ---------   >>
3307 0111 LIT  0003      )
330B 142C OVER           )
330D 17BE =              )------   <<
330F 07FD $IF  3319      )
3313 325D THIRD
3315 0810 $ELSE  331B ---------   >>
3319 326F WHO-KNOWS?
331B 13BB DROP ----------------   ENDCASE
331D 01C8 $;
OK
```

FIGURE 4

```
( FORTH CONTROL STRUCTURES ) BASE @ HEX
: !CADR WPARAM - , ;
: NOT
   IF 0 ELSE 1 THEN ;
: WHILE
   HERE ;  IMP WHILE
: PERFORM
   ' DUP !CADR
   ' <R !CADR ' $IF !CADR
   HERE 0 , ;     IMP PERFORM
: ENDWHILE
   HERE SWAP ! ' R> !CADR
   ' NOT !CADR ' $IF !CADR , ;
IMP ENDWHILE
BASE !   :S


( FORTH CONTROL STRUCTURES ) BASE @ HEX
: UNTIL ;    IMP UNTIL
: CASE 0 0 ;   IMP CASE
: SEL 0 -1 ;    IMP SEL
: COND 0 -2 ;    IMP COND  ( DO CONDITIONAL BRANCH )
: >>
   ' $ELSE !CADR 0 , HERE
   SWAP ! HERE 2 - SWAP ;   IMP >>
: ENDSEL DROP ( CASE#/FLAG )
   HERE
   WHILE OVER PERFORM
       DUP ROT ! ENDWHILE
   2DROP ' DROP !CADR ;
: ENDCASE ENDSEL ;              : ENDCOND SEL :
IMP ENDSEL   IMP ENDCASE   IMP ENDCOND
BASE !   :S


( FORTH CONTROL STRUCTURES ) BASE @ HEX
: WHEN
   DUP -2 =
   IF ' OVER !CADR
      ' = !CADR
   THEN
   ' $IF !CADR
   HERE 0 , ;
: << DUP 0< IF
   DUP -2 = IF ' DUP !CADR THEN ( COND )
   ELSE ' LIT !CADR 1+ DUP , WHEN THEN ;
IMP <<   IMP WHEN
: OTHERWISE ;   IMP OTHERWISE
BASE !   ;S
```

FIGURE 5

MEETINGS

NORTHERN CALIFORNIA

9/27/80

Dave Lion announced availablility of his 6800 assembler in FORTH occupying 1.5 Kbytes of 4 screens.

Tom Zimmer annonced availability of his Tiny Pascal in FORTH; Ragsdale again lauded Tom's effort as a benchmark (cf., MEETING REPORT, FD vol. 11 No. 3, p. 59).

Martin Schaaf announced committee formation for specifying a FORTH machine's hardware.

Henry Laxen of ORTHOCODE Corp. made freely available a FORTH "WORDSTAR"-styled Editor and announced sale of GOING FORTH, the tutorial package on 8" disk by CREATIVE SOLUTIONS.

Eric Welch, the FORTH Programming Team Manager for FRIENDS-AMIS' pocket computer project, gave an in-depth description of his job. A philosophy of team organization and control was graphed and an iterative planning strategy delineated. Some problems encountered and solved by this management strategy included:

- wheel-reinvention, duplication and redundancy prevention
- tool development (much effort was spent on tracers, patches, simulators, target compiler, breakpoints and documentation and its maintenance)
- style adherence (readability and maintainability) in development and documentation
- programming environment (which, in FORTH, is relatively worse due to newness and inexperience)--here the solution entails the project manager's close involvement and intense team interaction
- accountability of time spent at each level of the plan

How to form a FIG Chapter:

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

2. Send to FIG in San Carlos, CA 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).

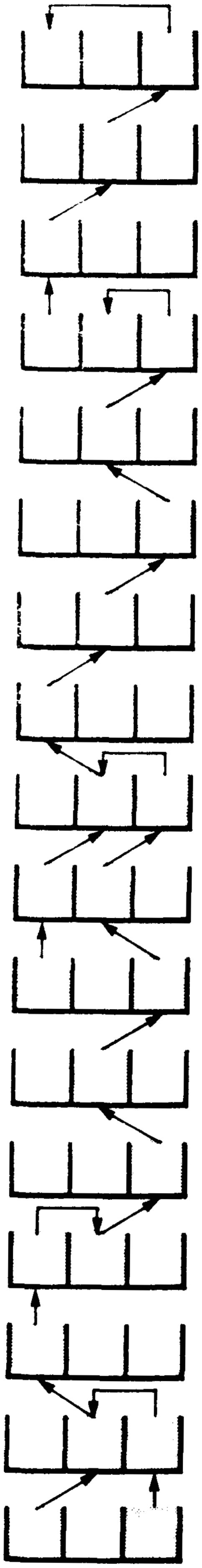
3. FIG will print, address and mail to members with the ZIP's you want from San Carlos, CA.

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 Saturday	FIG Monthly Meeting, 1:00 p.m., at Liberty House Department Store, Hayward, CA. FORML Workshop at 10:00 a.m.
Southern California 4th Saturday	FIG Meeting, 11:00 a.m. Allstate Savings, 8800 So. Sepulveda, L.A. Call Phillip Wass, (213) 649-1428.
FIGGRAPH 11/15/80 12/13/80	FORTH for computer graphics. 2:00 p.m. at Stanford Medical School, #M-112 at Palo Alto, CA.
Massachusetts 3rd Wednesday	MMSFORTH Users Group, 7:00 p.m., Cochituate, MA. Call Dick Miller at (617) 653-6136 for site.
San Diego Thursdays	FIG Meeting, 12:00 noon. Call Guy Kelly at (714) 268-3100 x 4784 for site.
Seattle Various times	Contact Chuck Pliske or Dwight Vandenburg at (206) 542-8370.
Potomac Various times	Contact Paul van der Eijk at (703) 354-7443 or Joel Shprentz at (703) 437-9218.
Texas Various times	Contact Jeff Lewis at (713) 729-3320 or John Earls at (214) 661-2928 or Dwayne Gustaus at (817) 387-6976. John Hastings (512) 835-1918
Arizona Various times	Contact Dick Wilson at (602) 277-6611 x 3257.
Oregon Various times	Contact Ed Krammerer at (503) 644-2688.
New York Various times	Contact Tom Jung at (212) 746-4062.
Detroit Various times	Contact Dean Vieau at (313) 493-5105.
Japan Various times	Contact Mr. Okada, President, ASR Corp. Int'l, 3-15-8, Nishi-Shimbashi Manato-ku, Tokyo, Japan.

Publishers Note:

Please send notes (and reports) about your meetings.



FORTH DIMENSIONS

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

Volume II
Number 5
Price $2.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No. 5 January/February 1981

Publisher Roy C. Martens

Guest Editor S B Bassett

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL 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 $12.00 per year ($24.00 overseas 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 for 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, although our membership of 2,400 is worldwide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

EDITOR'S COLUMN

Alan Taylor, in his speech at the FORTH Convention banquet, pointed out that FORTH is an incredibly powerful tool, and not precisely a language, in the traditional sense.

FORTH works close to the "naked machine" and yet is as general and powerful as many High Level Languages. This makes FORTH the perfect language for writing compilers or pseudo-compilers for other languages.

If we were to produce a compiler for ANSI Standard COBOL, for instance, COBOL would run on any machine which had FORTH running—which is easy!

A major software house is presently writing an Ada compiler, and expects to have it out in 1983.

I am willing to bet that a working FORTH compiler for Ada could be written in half that time—does anyone want to take me up on the bet?

S. B. Bassett

PUBLISHER'S COLUMN

1980 retropective: FORTH DIMENSIONS has completed a whole year with the new format and a number of people think it gets better with each issue. FORTH Interest Group has grown from 647 members on July 1st to over 2,400, thanks to *Byte*, *EE Times*, *InfoWorld*, *ComputerWorld* and other publications. The FORML Conference and FIG Convention were great successes. Many new chapters are being formed, how about you forming one? FORTH vendors are increasing almost as fast as the membership. FORTH is being implemented on more and more machines and applications are beginning to roll out. It's been a great year for us and we hope for you.

1981 perspective: FORTH DIMENSIONS will get better, we will have paid guest editors for each issue. A number of new publications and other FIG items will be introduced (see order form). A number of mailings will be done to FIG members about products available (if you *don't* want to receive these mailings, please drop us a note). We'll be doing more publicity to trade magazines and at computer shows. We need more aticles for FORTH DIMENSIONS, send yours in. Happy New Year!

Roy Martens

A NEW SYNTAX FOR DEFINING DEFINING WORDS

William F. Ragsdale

ABSTRACT

The computer language FORTH utilizes a syntax that is generally context free (i.e., postfix, or reverse Polish). However, deviations from this principle are noted in the syntax for words that themselves define words. This paper presents an altered form, which improves clarity of expression, and generalizes the construction for compilers which generate FORTH systems (meta-FORTH).

BACKGROUND

Compilation of FORTH programs consists of adding to memory a sequence of numerical values (addresses) corresponding to source text (words). This period is called compile-time. These values, called compilation addresses, are later interpreted by the address interpreter (at their run-time). They specify actual machine code which is ultimately executed, under control of the address interpreter.

1. FORTH source syntax is close to FORTH object code.
2. Traditional computer languages require significant processing to convert their syntax to object code. The syntax conversion is specified using Backus-Naur statements or "rail-road-track" diagrams of N. Wirth.
3. The traditional compiler's conversion adds complexity to the compiler, increases complexity and compiler size. It also masks the results from the user so that the use can't see or control the object code FORTH reduces complexity by requiring the user to write in a direct, simple syntax.

The espoused benefits are:

1. The programmer directly controls program flow. Inefficiency should be more apparent to the programmer.
2. The compiler is simpler, smaller, and more time efficient.
3. Compiler functions may be added by the programmer consistent with those previously in the system.

The arguments against having to write in this form include:

1. God created infix notation. If not so, why did we learn it as children? God doesn't lie to little children.
2. Languages are created by compiler writers, not compiler users. Therefore, let these brilliant sources have a larger part of the pie (read headaches for pie).

For completeness, it should be noted that program branching requires reference to addresses not at the point of compilation. The compiling words of FORTH (DO, UNTIL, IF, etc.) use the compile time stack to hold interim addresses which specify such branching. The nesting of conditionals keeps this process simple and efficient, and obiviates the need for backtracking or looking ahead in the source text.

A PROBLEM

Three exceptions to a context free expression exist in FORTH as generally used and formalized in FORTH-79:

1. The word IMMEDIATE sets the precedence flag of the most recent definition in the CURRENT vocabulary. Location of this bit is done via a variable/vocabulary pointer pointing backward in memory an unknown amount. If selection of the CURRENT vocabulary has been altered, the wrong definition is made immediate.

2. Defining words create a dictionary word header, but some other word backtracks in the object code to change the execution procedure assigned to each word created. E.g.,

```
: C-ARRAY CREATE ALLOT DOES> + ;

10 C-ARRAY DEMO
```

   The word DEMO is created by CREATE as a variable and is proximately altered by DOES> to execute with a much different role in making a run-time address calculation.

3. ; and END-CODE make available for use each correctly compiled definition. This is often determined from an alterable pointer, sensitive to the vocabulary specified as CURRENT.

To display these cases together:

```
: WWW . . . ;
: XXX . . . ; IMMEDIATE
: YYY . . . ;CODE . . . END-CODE
: ZZZ . CREATE . . . DOES> . . . ;
```

Each of the above words is quite different in function and execution, yet they were all defined by : . The user must analyze the contents of each definition to determine what type of word it is (i.e., colon-definition, compiling-word, code-definer, or high-level-definer). Because of these varied forms, the glossary definition of : is only partly complete. The other variations on : must be discovered as they occur.

Creation and use of the above types is complicated in that the resulting functions are dependent on words following within and outside (!) each definition. As new words are defined by CREATE and assigned execution code by DOES> and ;CODE , the compiling function must backtrack by implicit pointers to alter previously generated word headers.

Added commentary is appropriate for item 2, above. It is a general characteristic of FORTH that a word's function may be uniquely determined by the contents of its code field. This field points to the actual machine code which executes for this word. Common classes of words which are consistent include: variables, constants, vocabularies.

This is not the case with defining words. These words all have the same code field contents as any other colon-definition which indicates that they execute interpretively until the concluding ; . But actually the intervening DOES> or ;CODE terminates execution and alters the specification for the execution of the word being defined. Philosophically, it appears that this is the grossest form of context sensitivity of any language, due to the generality and power of the construct.

But this power and generality contains its own downfall. It increases the complexity of comprehension and complexity of compilation. When a novice competently begins to use DOES> and ;CODE he has come of age in FORTH.

THE GOAL

The goal of the proposed new technique is a uniform expression of source text that may be compiled for resident RAM execution, resident ROM execution or target execution (from a binary image compiled to disk for later execution). To enable this uniformity, a context free expression is used.

THE PROPOSAL

The proposed syntax for defining words uses only the compile time stack (or dedicated pointers), generating object code and word headers linearly ahead. Each word type has a unique defining word so that no later modification of a word definition need be made. A meta-defining word is proposed which makes all defining words. Each defining word is obvious because each, itself, is created by this "meta-definer".

This meta-definer is BUILDS> . This name is an old friend to some, since it was the name of the word previously used where CREATE is specified by FORTH-79. This word still has its old role of building words which themselves build words, but is used in a more obvious fashion.

Here is an example, written in FORTH-79 for a word which creates singly dimensioned byte arrays:

```
: C-ARRAY   CREATE  ALLOT
            DOES>    +   ;
```

It would be used in the form:

```
10  C-ARRAY  DEMO
```

to make an array named DEMO with space for 10 bytes. When DEMO executes it takes an offset from the stack and returns the sum of the allotted storage base address plus the offset.

Using the proposed new meta-definer BUILDS> the creation of C-ARRAY is:

```
DOES> + ;
  (the run-time part)
BUILDS> C-ARRAY  ALLOT ;
  (the compile-time part)
```

And is used:

```
10  C-ARRAY  DEMO
  just as above.
```

It should be noted that the impact of the use of BUILDS> is only in defining defining words. Later use of such defined words would be as presently conventional.

THE NEW SYNTAX

Here is a summary of the defining word syntax that appears at the application level. Note that these examples are very close to what we commonly use in FORTH-79.

```
: <name> . . . . ;
```

Define a non-immediate word which executes by the interpretation of sequential compilation addresses.

NOW <name> . . . . ;

Define an immediate word which executes by the interpretation of sequential compilation addresses, and will execute when encountered during compilation.

CREATE <name>

As in FORTH-79.

n CONSTANT <name>

As in FORTH-79.

VARIABLE <name>

As in FORTH-79.

VOCABULARY <name>

As in FORTH-79, but each defined vocabulary is immediate.

When the programmer creates new word types, a significantly different syntax is used, as compared to FORTH-79.

DOES> . . . . ;

Begin the nameless run-time high-level code for words to be defined by <name>.

BUILDS> <name> . . . . ;

<name> <namex>

Define <name> which, when later executed will itself create a word definition. The code after <name> executes after creating the new dictionary header for <namex> to aid parameter storage. The previous run-time code is assigned to each word <namex> created by <name>.

When new classes of words are created with their run-time execution expressed by machine code, their defining word is created thusly:

CODE> . . . . END-CODE

Begin the nameless run-time machine code for words to be defined by <name>.

BUILDS> <name> . . . . ;

<name> <namex>

Define <name> which, when later executed will itself create a word definition. The code after <name> executes after creating the new dictionary header for <namex> to aid parameter storage. The previous run-time code is assigned to each word <namex> created by <name>.

THE METHOD

We will follow the method of the honey bee. To propagate the colony the bees need a queen bee. An ordinary bee is fed special hormones to become a queen bee. By regulating this process the colony regulates its growth.

Our queen bee will be BUILDS> . It is originally created as a colon definition. Then it is converted into a new type of word that creates words which always create. This form uses parameters to create a dictionary entry and then passes control to the users code which specifies completion of the entry.

We will break the CREATE DOES> construct into two parts. The creating part will be called BUILDS> with the right pointer emphasizing that the following word 'builds' other words. BUILDS> is the meta-defining word since it is the source of all defining words. It must be emphasized that the word creating function is inherent in any word created by BUILDS> , and need not be additionally specified.

The execution procedure is begun by CODE> (for words with a machine code execution) or by DOES> (for words with a high-level execution). Coupling from these two words is accomplished by passing an address and bit mask from DOES> or CODE> to BUILDS> .

The precedence of a word traditionally is set by declaring each such word as IMMEDIATE . In the new form, this is declared for the defining word, not for each word as defined. By executing IMMEDIATE after the CODE> or DOES> part, but before the BUILDS> part, the bit mask on the stack is altered to the immediate form. This mask is applied to all words as later defined, so all will be immediate.

Usually colon-definitions and code definitions are created 'smudged' so that they will not be found during a dictionary search. When successfully compiled, the smudge bit is reset, making the word available for use. Other words are much less susceptible to errors of compilation, and so are created un-smudged. The smudge function is not generally manipulated by the user but completed by ; or END-CODE. The smudge bit is contained in the header count byte.

By executing SMUDGE after the CODE> or DOES> part, but before the BUILDS> part, every word later created will be created smudged. It is a system choice how the un-smudging is performed. It is suggested that a pointer uniquely specify the current smudged bit address.

Some systems achieve the same result by selectively linking words into the dictionary. In this case the selective linking is done by the defining part of BUILDS> as selected by the bit mask associated with each defining word.

A major problem exists for meta-compilation (target-compilation) of new defining words. The compile-time portion must know the run-time compilation address corresponding to each word type. Several methods are currently used. In all cases the syntax is a deviation from the usual version suitable for testing on a resident system. Part of the art of target compilation is knowing how to alter resident defining words to operate in the target compilation situation.

The programmer may declare byte counts to allocate memory space and later re-origin compilation to fill in code fragments. Other techniques consist of compiling the full structure and then passing address locators to previously defined words. In poly-FORTH, dual definitions are used. The target compilation definition of our C-ARRAY example is:

```
: C-ARRAY CREATE ALLOT ;CODE FORTH
: C-ARRAY CREATE ALLOT DOES> +  ;
```

It is an exercise in ingenuity to determine which parts of the above code end up in the target system, and which are added to the host compiler.

Here is a summary of the meta-compiling of our example:

```
DOES>  +  ;
BUILDS>  C-ARRAY  ALLOT  ;
10  C-ARRAY  DEMO
```

First the DOES> compiles <does> + ; into the target system and passes the locating parameters to BUILDS> . <does> is an in-line code vector to machine code.

Then the BUILDS> compiles C-ARRAY ALLOT ; into the target system with the proper object locators for the DOES> part and then places another copy of C-ARRAY ALLOT ; into the resident compiler so that C-ARRAYs may be immediately defined for the target system.

Finally, the C-ARRAY in the host system executes to place a definition for DEMO into the target system, locate the address of DEMO for later compilation, and finally ALLOT ; makes the target memory allocation and concludes the target definition.

The only source changes anticipated are the occasional explicit change of vocabulary to correctly select (during target compilation) words which affect the application memory. Again, this is only done for selected defining words.

The key to this method is that the run-time portion is known before the compile-time portion, and the creation of defining words is done uniformly, linearly ahead.

CONCLUSION

A complete implementation of these concepts follows. A six word glossary expands the explanations given above. This implementation is written in FORTH-79, with system dependent words taken from fig-FORTH. The source of each word is identified in the Appendix.

This construction for BUILDS> is offered as a method to regularize the structure of FORTH at the defining word level. Its success will be judged by either usage or the stimulation of other methods for this purpose.

GLOSSARY

BUILDS>   addr mask ---

A defining word used in the form:

BUILDS> <name>. . . . ;

to define a defining word <name> . The address and mask (left by either DOES> or CODE>) are placed into the definition of <name> to specify the header structure for all words created by <name> and locate the execution procedure assigned by <name> . The text between <name> and ; is compiled to complete the definition of <name> .

When <name> executes in the form:

<name> <namex>

it generates a dictionary entry for <namex> and then executes the code following <name> to finish compilation of <namex> .

When <namex> executes, it executes the code in the DOES> or CODE> part preceding <name> . Refer to DOES> or CODE> .

CODE>   --- addr mask

Used in the form:

CODE>..(assembly text)..END-CODE

to begin the nameless compilation of a sequence of assembler code text. The address and mask left locate this sequence for BUILDS> . The mask contains the precedence and smudge bits and may be altered by IMMEDIATE and/or SMUDGE while still on the stack, before being compiled by BUILDS> .

When a word with a CODE> part ultimately executes, it executes the code between CODE> and END-CODE, at a machine code level. Execution must ultimately be returned to the address interpreter NEXT .

DOES> --- addr mask

Used in the form:

DOES> . . . . ;

to begin the nameless compilation of a sequence of high-level code. The address and mask left locate this sequence for BUILDS> . The mask contains the precedence and smudge bits and may be altered by IMMEDIATE and/or SMUDGE while still on the stack, before being compiled by BUILDS> .

When a word with a DOES> part ultimately executes, it executes the code following DOES> with its own parameter field address automatically placed on the stack.

IMMEDIATE addr mask --- addr mask

Set the precedence bit in the mask to indicate that all words later defined by the defining word being defined will always execute when encountered.

Immediate words are aids to compilation, such as:

IF BEGIN DO ." etc.

NOW

A defining word used in the form:

NOW <name> . . . . ;

to define <name> in the fashion of : , but in the immediate form. That is, <name> will execute even when encountered during compilation.

SMUDGE addr mask --- addr mask

Set the smudge bit in the mask to indicate that all words defined by the defining word being defined will begin in the 'smudged' condition. This condition prevents a word from being found in a dictionary search until un-smudged at the completion of correct compilation.

APPENDIX

The example implementation of the new BUILDS> is written in FORTH-79 running on a 6502 processor. When system dependencies occur, the fig-FORTH methods were used regarding error control and dictionary header structure. Here is a tabulation of the pedigree of each word (its origin) used in this application.

Numbers indicate a standard definition from FORTH-79, fig indicates the definition from fig-FORTH. Assembler words are from a 6502 assembler.

!	122	AGAIN	fig	LOOP	124
!CSP	fig	ALLOT	154	MIN	127
'	171	ASSEMBLER	fig	OR	223
(	122	BL	fig	OVER	170
+	121	C@	156	QUIT	211
+!	157	C,	fig	ROT	212
,	143	CFA	fig	SMUDGE	fig
-	134	COMPILE	146	SWAP	230
1+	107	CONTEXT	151	TOGGLE	fig
1-	105	COUNT	159	VARIABLE	fig
2+	135	CSP	fig	VOC-LINK	fig
:	116	CURRENT	137	WORD	fig
;	196	DO	142	[	125
=	123	DP	fig	[COMPILE]	179
>R	200	DUP	205	]	126
?CSP	fig	HERE	188		
@	199	I	136		
		LOAD	202		

```
SCR # 6
( Adopt form of FORTH-79                              WFR-80NOV08 )
: CREATE   0  VARIABLE  -2  ALLOT  ;
: 1-    1   -  ;                       : 2-   2  -  ;
: WORD   WORD  HERE  ;
: 'SMUDGE   SMUDGE  ;   ( rename for access to old version )
: END-CODE    ASSEMBLER  [COMPILE]  C;  ;

( ********* BEGINNING OF THE NEW  BUILDS>  PACKAGE ***********)
79-STANDARD  ( but this is not a standard program )
HEX
: 2DUP    OVER  OVER  ;
: THRU    1+  SWAP  DO  I  LOAD  LOOP  ;
: IMMEDIATE     40   OR   ;          ( next word defines immediates )
: SMUDGE        20   OR   ;             ( next word defines smudged )

DECIMAL    7 11 THRU    DECIMAL

SCR # 7
( META-definitions of DOES> and CODE>                 WFR-80NOV08 )
CREATE  <DOES>              ASSEMBLER  HEX
   DEX,  DEX,                            ( make room for pfa value )
   CLC,  2 # LDA,  W    ADC,  BOT    STA,
                TYA,  W 1+ ADC,  BOT 1+ STA,       ( copy the pfa )
   SEC,  PLA,  1 # SBC,  W STA,
         PLA,  0 # SBC,  W 1+ STA,  ( ready for hi-level call )
   ' QUIT CFA @ JMP,    ( make hi-level call for DOES> )

: DOES>      ( run time of META word  --- cfa, count byte mask )
   HERE  80  ( leave locators )  !CSP  'SMUDGE ( compensate ; )
   ]  ( begin compiling )  20  C,  COMPILE  [ <DOES> , ]   ;

: CODE>      ( run time of META word  --- cfa, count byte mask )
   HERE  80  ( leave locators )  !CSP  'SMUDGE
   [COMPILE]  ASSEMBLER  ;

SCR # 8
( META-definition of BUILDS>                          WFR-80NOV08 )
DOES>          ( run-time for meta-BUILDS> which makes headers )
   COUNT  ( pfa+1, count mask )
   BL  WORD  DUP  C@  1+  ALLOT  ( the name )
   DP  C@  OFD  =  ALLOT  ( for 6502 only )
   DUP  ROT  TOGGLE  HERE  1-  80  TOGGLE  ( name marker bits )
   CURRENT  @  @  ,  CURRENT  @  !  ( link into vocabulary )
   DUP  @  ,  ( lay down code field )
   2+  >R  ( resume the BUILDS> word  )  ;

SMUDGE
: BUILDS>                                  ( begins defining words )
   [ -2 ALLOT OVER ,  ( change cfa to above DOES> )  4 CSP +!
         C,  ,  ( lay down count mask, then cfa )  ]
     ?CSP  C,  ,  !CSP  CURRENT  @  CONTEXT  !   ]  ;


SCR # 9
(   CODE  ;  and  NOW                                 WFR-80NOV08 )
CODE>      END-CODE       SMUDGE       ( no execution procedure )
BUILDS> CODE                     ( create a smudged code definition )
        HERE  DUP  2-  !  [COMPILE]  ASSEMBLER  !CSP  ;

CODE>                                      ( for colon-definitions )
   IP 1+ LDA,  PHA,  IP LDA,  PHA,  CLC,  W LDA,  2 # ADC,
   IP STA,  TYA,  W 1+ ADC,  IP 1+ STA,  NEXT JMP,   END-CODE

2DUP  SMUDGE             -4  CSP  +!   ( use these params twice  )
BUILDS>  :              ( create new colon-definitions till ';'  )
           !CSP  CURRENT  @  CONTEXT  !  ]  ;

IMMEDIATE  SMUDGE
BUILDS>  NOW                ( creates immediate colon-definitions )
           [ ' : 3 + (  share code within ':' )  1 ]  AGAIN  ;

SCR # 10
(  VARIABLE  CREATE  and  CONSTANT                    WFR-80NOV08 )
CODE>                                              ( for variables )
   CLC,  W LDA,  2 #  ADC,   PHA,
             TYA,  W 1+ ADC,  PUSH JMP,    END-CODE

2DUP                  -4   CSP  +!           ( share run-time code )
BUILDS> CREATE  ;                         ( general purpose creator )

BUILDS> VARIABLE  0 , ;  ( create a variable, not initialized  )


CODE>                                              ( for constants )
   2 # LDY,  W )Y LDA,  PHA,
        INY,  W )Y LDA,  PUSH JMP,   END-CODE

BUILDS>  CONSTANT  ,  ;  ( create a constant, value from stack )

SCR # 11
(  VOCABULARY,  ARRAY                                 WFR-80NOV08 )
DOES>   2+  CONTEXT  !  ;

IMMEDIATE        ( note that all vocabularies will be immediate )
BUILDS>  VOCABULARY
       A081  ,  CURRENT  @  CFA  ,
       HERE  VOC-LINK  @  ,  VOC-LINK  !  ;

VOCABULARY A-TRIAL


       ( one dimensional byte array, confined within allocation )
DOES>    COUNT  ROT  MIN  +  ;
BUILDS>   C-ARRAY  DUP  1-  C,  ALLOT  ;

10  C-ARRAY FOR-TEST
```

FIG GROUPS

Standards--Bill Ragsdale, c/o fig, P.O. Box 1105, San Carlos, CA 94070.

FORML--Kim Harris, P.O. Box 51351, Palo Alto, CA 94303.

8080 Renovation Project--cleaning up the figFORTH 8080 implementation --Terry Holmes, c/o fig, P.O. Box 1105, San Carlos.

figGRAPH--Howard Pearlmutter, c/o fig P.O. Box.

figSLICE--The FORTH Machine, to be built with bit slice technology--Martin Schaaf, 202 Palisades Dr., Daly City, CA 94015.

figTUTOR--how to teach FORTH to new people--forming--Sam Bassett, c/o fig P.O. Box.

HELP!! MAYDAY!!

The Editors, not being "old FORTH hands", need experienced LOCAL help in testing submitted programs.

Diversity of systems (fig or not) and terminals much appreciated.

Reply to fig P.O. Box, please!

INPUT NUMBER WORD SET

Robert E. Patten

Purpose

The FORTH primitives <# # #S $ SIGN #> allow generalized numeric output. This paper presents a generalized method for numeric input.

Method

This word set, as implemented, will convert a word placed at HERE, terminated with a trailing blank, to a double integer on the data stack. The type of input converted is available in the variable TRAIT.

This word set will allow extentions to include other number and data types (i.e., floating point, triple precision numbers, and simple string parsers).

Most words in this word set expect a flag on the data stack and leave a flag indicating success or failure of the conversion or test performed. A true flag indicates success. The word CHR is an exception. CHR replaces the flag with a character from the word at HERE. If the last conversion or test was a failure, CHR leaves the same character. If the last operation was a success, CHR leaves the next character on the data stack.

The defining word N: may be used to create a word to convert the word at HERE to a double integer. A successful conversion will leave a double integer and a true flag. A failure will leave only a false flag. If words defined by N: are used to define a word created by UNTIL: then this new word will, when executed, try each N: created word on the word at HERE until one is successful, leaving only a double integer on the stack. If none of the words are successful then nothing is left on the data stack. This outcome is not acceptable because no number was put on the data stack. Because of this, the last word in the UNTIL: defined word should cause a "Word not defined" error.

INPUT NUMBER WORD SET

```
(<N)        --- flag  d flag
```

Leave a plus sign, a double number zero, and a true flag on the data stack in preparation for number conversion.

```
(N>)        sign d flag --- d true
                        --- false
```

If flag is true apply sign to double number and leave number and true flag else leave only false flag.

```
.N          dl flag --- d2 flag
```

Substituting zero for blank, convert CHR into double number beneath leaving true flag if ok, else leave false flag.

```
>CHR        --- addr
```

Leave the address of a variable which contains a pointer to the last character fetched by CHR.

```
?,NNNS      dl flag --- d2 flag
```

Allow groups of comma and three digits to be converted to double number beneath. If no comma return true flag. If no three digits following the comma then return false flag.

```
?.          flag --- flag
```

If CHR is a period then set DPL to zero and leave true flag else leave false flag.

?ASCII flag --- flag

If CHR equals character following ?ASCII then leave true flag else leave false flat.

?BOTH flag --- flag

If flag is true and CHR is a blank then leave a true flag else leave a false flag.

?END flag --- flag

If CHR is a blank then leave a true flag else leave a false flag.

?SIGN false d flag --- sign d flag

If CHR equals - change false flag to a true flag and leave true flag on top else leave false flag on top.

?SKIP flag --- flag

Make flag true. Used to skip past character if flag was false.

ASCII --- char

Place following character on data stack as a number.

CHR flag --- char

Add flag to >CHR and fetch character at >CHR to data stack.

N: A definining word used in the form:

```
N: <name> . . . ;
        --- d true
        --- false
```

Convert word at here leaving double number and a true flag on the data stack. If word does not convert leave only a false flag on the data stack.

N dl flag --- d2 flag

Convert digit at CHR into double number beneath. If successful leave true flag else a false flag.

NNN dl flag --- d2 flag

Do three N leaving true flag if successful else false flag.

NS dl flag --- d2 flag

Do N until failure, leaving a true flag on top of data stack with >CHR pointing to last character accepted

(N) dl digit --- d2

Convert binary digit into number beneath.

REQUIRED flag --- flag

If flag is false exit this word leaving false flag. If flag is true leave true flag and continue.

TRAIT --- addr

A variable containing the word count from the last UNTIL: defined word.

UNTIL:

A defining word used in the form:

```
UNTIL: <name> . . . ;
```

Words created by UNTIL: are like colon-definitions except the run time function is to execute words in the definition until there is a true flag on the data stack, then exit the word leaving the word count of the words executed in the variable TRAIT.

```
( ASCII TO BINARY WORD SET REP )  BASE @ DECIMAL
0 VARIABLE >CHR
: (<N) ( --- ff d tf )   0 0.0 1   -1 DPL ! HERE          >CHR ! ;

: CHR   ( f --- character )  >CHR +! >CHR @   C@;

: (N>) ( f d1 f --- d2 tf   good number
                --- ff      bad number  )
  >R ROT IF DMINUS THEN R>  IF 1 ELSE DROP DROP 0 THEN ;

: COLON  ( Build : def. with security.)
 ?EXEC !CSP  CURRENT @ CONTEXT !  <BUILDS ]  SMUDGE ;

: N:  ( --- d tf / --- ff )
    COLON   DOES>              ' (N>) >R ( EXECUTE AFTER )
    >R  ( EXECUTING PARAMETER FIELD )  (<n) ( SET UP STACK ) ;

 0 VARIABLE TRAIT
: UNTIL:     COLON DOES> >R  -1 TRAIT !
   BEGIN    1 TRAIT +!   R @ EXECUTE  R> 2+ >R UNTIL
   ( EXECUTE WORDS UNTIL TRUE)    R> DROP  ( THEN EXIT. ) ;

: (N) ( d1 digit --- d2) SWAP BASE @ U* DROP  ROT BASE @ U* D+
      DPL @ 1+  IF 1 DPL +! THEN ;

: N ( d1 f --- d2 f ) CHR BASE @ DIGIT  IF (N) 1 ELSE 0 THEN ;

: NNN ( d1 f --- d2 f )  N N N ;

: NS ( d1 f --- d2 tf ) N BEGIN  DUP WHILE N REPEAT
      0= -1  >CHR +!  ;

: REQUIRED ( f --- f : if false exit else continue )
      DUP 0=  IF R> DROP THEN ;
: ASCII       BL WORD HERE 1+ C@   [COMPILE] LITERAL ; IMMEDIATE
: ?ASCII ( f --- f )   COMPILE CHR [COMPILE] ASCII
   COMPILE = ;  IMMEDIATE
: ?SIGN ( ff d f --- f d f )  ?ASCII - DUP
  IF >R ROT 0= ROT ROT R> THEN ;
: ?,NNNS   ( d1 f --- d2 f )   BEGIN ?ASCII ,  DUP
    WHILE  NNN REQUIRED REPEAT  0=   -1 >CHR +!  ;
: ?. ( f --- f ) ?ASCII .   DUP  IF 0 DPL ! THEN ;
: ?SKIP  ( f -- tf ) DROP 1 ;
: ?END ( f --- f )  CHR BL = ;
: ?BOTH ( f --- f ) DUP IF ?END    THEN ;
: .N ( d1 f --- d2 f    AUTOSCALE)
   N DUP 0= IF ?END IF 0 (N) 1  ELSE 0 THEN THEN ;

N: INTEGER ?SIGN NS ?,NNS REQUIRED  ?END ;
N: REAL ?SIGN NNN ?,NNNS REQUIRED ?. NS ?END   ;

 : :NN  ?ASCII : REQUIRED   6 BASE ! N REQUIRED  DECIMAL N ;
N:  TIME  ( HH:MM:SS ) DECIMAL   N N DROP OVER  24 < 0= 0=
    REQUIRED   :NN REQUIRED :NN  ?BOTH REQUIRED 0 DPL ! ;

N: SSN DECIMAL NNN REQUIRED ?ASCII - N N REQUIRED ?ASCII - NNN N
 ?BOTH REQUIRED 0 DPL ! ;         N: ZERO ;
N: AREA-CODE  ?ASCII ( REQUIRED NNN REQUIRED ?ASCII ) ?BOTH ;
N: PHONE NNN REQUIRED ?ASCII - REQUIRED NNN N ?BOTH REQUIRED
   0 DPL ! ;
N: DOLLAR DECIMAL  ?ASCII $ REQUIRED ?SIGN NNN ?,NNNS ?. .N .N
    0 DPL ! ;

: (BAD) 0 ERROR ;

UNTIL: (NUMBER) INTEGER REAL DOLLAR TIME SSN PHONE AREA-CODE
   (BAD)  ;

: NUMBER DROP BASE @ >R  (NUMBER) R> BASE ! ;

' NUMBER CFA ' INTERPRET 36 + !

BASE ! ;S


( TEST (NUMBER)                  14 LOAD
: GET-NUMBER     QUERY BL WORD   HERE  NUMBER D. TRAIT ? DPL ? ;

: NUMBER-TEST    BEGIN CR GET-NUMBER ?TERMINAL  UNTIL ;

: X IF D. DPL ? . ELSE ." BAD " THEN ;

: TEST-N: [COMPILE] ' QUERY BL WORD CFA EXECUTE X ;
: <N (<N) QUERY BL WORD ;
: N> (N>) X .S ;
                          ;S
```

ATARI DISKETTE

Diskette and documentation for fig-FORTH on ATARI computers. Runs on one disk drive and 16K RAM. Has full screen editor and extensions. $50.00

Bob Gonsalves
c/o Pink Noise Studios
1411 Center Street
Oakland, CA 94607

NEW PRODUCTS

FORTH for OSI

by Forth-Gear

Forth-Gear is pleased to announce the release of a complete FORTH software package for several models of Ohio Scientific Instruments computers. The Forth Interest Group model language runs under OSI's Disk Operating System OS65D-3.2, but high level FORTH DOS words are implemeted in FORTH for full compatibility with fig-standard extensions. A line editor is included for the creation and disk storage of FORTH programs. A 6502 assembler permits the use of machine code routines as FORTH definitions. The editor and assembler may both be extended by the creation of new definitions in high level FORTH.

Included with the package are several utility programs in FORTH, including a RAM Dump, video graphics, data disk initializer (may use all tracks except track zero), a sample machine code routine (screen clear), and a system disk optimizer.

Minimal system requirements are 24 Kilobytes of RAM and one disk drive. System attributes beyond the minimal requirements may be fully utilized by regenerating the system disk with the optimizer program. Two systems are currently available: The 5 1/4" disk version works on all C2-4P and C4 models. The 8" disk version works on all C2-8P, C8P, C2-OEM, and C3 models with either the polled keyboard or a serial terminal. Superboard, C1P, and C2 versions will be available very soon.

A single-user system consisting of a disk (specify size) and fifty page user manual is available from Consumer Computers, 8907 La Mesa Blvd., La Mesa, California 92041, for the introductory price of $69.95 prepaid. Telephone (714) 698-8088 9 to 5 PST.

STRUCTURED PROGRAMMING BY ADDING MODULES TO FORTH

Dewey Val Schorre

Structured programming is a strong point of FORTH, yet there is one language feature important for structured programming which is currently absent in FORTH. This feature is called a module in the programming language MODULA, and appears under other names in other languages, such as procedure in PASCAL. It can, however, be easily added by defining three one-line routines.

The names of these routines are: INTERNAL, EXTERNAL and MODULE. A module is a portion of a program between the words INTERNAL and MODULE. Definitions of constants, variables and routines which are local to the module are written between the words INTERNAL and EXTERNAL. Definitions which are to be used outside the module are written between the words EXTERNAL and MODULE.

One of the most common uses of modules is to create local variables for a routine. These variables are defined between INTERNAL and EXTERNAL. The routine which references them is defined between EXTERNAL and MODULE. Notice that this module feature is more general than the local variable feature of other programming languages, in that several routines can share local variables. Such sharing is important, not so much from the standpoint of saving space, but because it provides a means of communication between the routines.

If you have written any local routines between the words INTERNAL and EXTERNAL, then in order to debug them, you will have to delete the word INTERNAL and put a ;S before the word EXTERNAL. Since debugging in FORTH proceeds from the bottom up, once you have debugged these local routines, you will have no further need to refer to them from the console. They will only be referenced from the external part of the module. Modules can be nested to arbitrary depth. In other words, one module can be made local with respect to another by defining it between the words INTERNAL and EXTERNAL.

Now let's consider matters of style. The matching words INTERNAL, EXTERNAL and MODULE should all appear on the same screen. When modules are to be nested, one should not actually write the lower level module between the words INTERNAL and EXTERNAL, but should write a LOAD command that refers to the screen containing the lower level module. The screens of a FORTH program should be organized in a tree structure. The starting screen which you LOAD to compile the program is a module which LOAD's the next level modules.

Screens are much better for structured programming than the conventional character string file because they can be chained together in this tree structured manner. You will write a module for one program, and when you want to use it in another program, you don't have to edit it into the new program or add it to a library. All you have to do is to reference it with a LOAD command.

There is an efficiency advantage to the use of modules. One minor advantage is that compilation speed is improved because the dictionary that has to be searched is shorter. The more important advantage of saving dictionary space is not realized with this simple implementation, which changes a link in the dictionary. To save space, one would have to implement a dictionary that was separate from the compiled code. Moreover, this dictionary would not be a simple push-down stack, because the storage freed by the word MODULE is not the last information entered into the dictionary.

The words needed to define modules are as follows:

```
: INTERNAL  ( --> ADDR)  CURRENT @ @ ;
: EXTERNAL  ( --> ADDR)  HERE ;
: MODULE( ADDR1 ADDR2 --> )PFA LFA ! ;
```

FORML CONFERENCE

A Report on the
Second FORML Conference

The Second Conference of the Forth Modification Laboratory (FORML) was held over Thanksgiving, November 26 to 28, 1980, at the Asilomar Conference Center, Pacific Grove, California (some 120 miles south of San Francisco).

The weather was unseasonably beautiful, as the rainy season, normally starting in November, was late. Most conference attendees managed to find some free time to enjoy the beach and wooded areas.

With the way smoothed by a core crew who showed up Tuesday, the majority of participants arrived for lunch Wednesday, and launched right into a full schedule of technical sessions.

There were 65 conference attendees, with enough of them bringing family to raise the count to 96 people at Asilomar in connection with FORML.

The rooms were in scattered well-landscaped buildings. Meals were provided in a central dining building, and were generally praised. Thanksgiving noon dinner, a deluxe buffet meal, was a special treat.

The evening meetings, both Wednesday and Thursday, had formal technical sessions which evolved into quite open, informal, and productive discussions. The participants had to be persuaded to break up to move to the scheduled social gatherings over wine and cheese.

SUMMARY OF SESSIONS

The number of people presenting papers was so great (almost 40) that sessions were scheduled from Wednesday afternoon all the way to Friday afternoon. Topics of sessions, together with their chairmen, were:

FORTH-79 Standard
    Bill Ragsdale

Implementation Generalities
    Don Colburn

Implementation Specifics
    Dave Boulton

Concurrency
    Terry Holmes

FORTH Language Topics
    George Lyons

Other Languages
    Jon Spencer

MetaFORTH
    Armand Gamberra

Programming Methodology
    Eric Welch

Applications
    Hans Niewenhuijzen

In addition, Kim Harris, the Conference Chairman, opened the Conference with a welcome and a review of FORML-1, London, January 1980. Kim also closed the final session.

As one example of a conference paper, "Adding Modules to FORTH" by Dewey Val Schorre, gave a mechanism for setting up words which are local to a "module"--a sequence of FORTH code. His mechanism involves only three FORTH words, two of which already exist in FIG-FORTH. His novel but straightforward way of using these three simple words provides many of the benefits of VOCABULARY with less overhead, and by focusing on modularity, it can lead to clearer programs.

Another item of particular interest was George Lyons's paper on Entity Sets. His proposal is very economically implemented, and allows, at compile time, selection from lists of identically-named operators, such as @ ! + , based on data type.

These and other wonders will be published in the Proceedings of the Conference. This should be ready by the end of February, and will be sold by FIG.

LESS FORMAL OBSERVATIONS

At the Wednesday evening technical session an informal discussion on various topics included "Notes on the Evolution of a FORTH Programmer" by Charles Moore, in which he described how his own programming style had matured.

On the final day the question was brought up of whether FORTH was a programming language or a religion. The consensus was: Yes! In the same discussion the expression "born-again programmer" appeared. (It is in competition for catch-phrase of the year with "black-belt programmer", which was heard at the FIG Convention in San Mateo the following day.)

;s G. Maverick

LETTERS

J. E. Rickenbacker pointed out that the JMP ($xxFF) of the fig-FORTH inner interpreter does not work on a 6502.

That is right, but the fig-FORTH compiler automatically tests for this condition and avoids ending a CFA in FF.

The only problem occurs during initial installation when a hand assembly is required. Since 6502 assemblers, unlike FORTH, are inflexible you just have to sit there helpless watching them make the same dumb mistake at each new assembly and then add a correction when the assembler is finished. Since fig-FORTH has about 210 definitions, the chances are pretty good (about 210 out of 256) that a CFA will end in FF.

My advice would be to leave the patch in until the system is pretty well debugged and then install the jump indirect scheme of the fig-FORTH model. It would be a shame to permanently slow down the system unnecessarily because of an initial installation inconvenience which is primarily the fault of the inflexibility of the 6502 assembler.

As to Mr. Rickenbacker's query on a FORTH assembler vocabulary, he may find Programma International's version of APPLE-FORTH helpful. The system isn't FORTH, it is something like FORTH. However they have a FORTH-like assembler in their system which may be helpful. The op-codes have been analyzed for postfix operation, etc.

FORTH is beautiful.

Edgar H. Fey Jr.
La Grange, IL

LYONS' DEN

In the course of implementing the FIG model on my computer I have noticed that the word NOT is in the assembler vocabulary but not in the high level glossary. Instead 0= is used for logical negation in high level code. Defining NOT as a synonym for 0= in the main kernel glossary might be useful. Code would be a little more readable by distinguishing between the operations of testing whether a number on the stack from a mathematical formula is zero, and logically negating a boolean flag left on the stack by a relational operator, even though the code used to perform these two operations is the same. But a stronger need for a high level NOT occurs when floating point or other data types in addition to the standard integer type is implemented by a vocabulary containing redefinitions of the mathematical operators. In that case a new 0= would be defined to test, say, whether a floating point number were zero, and this new 0= could not be used for logical negation. Of course, the existing practice seems to be to define new operators with unique names such as F0= instead of redefining the kernel names, avoiding this problem. Also, a user can always add a synonymous NOT to the FORTH vocabulary before redefining 0= and the other operators in the vocabulary for a new data type. Once using NOT in code written in the terminology of the new vocabulary, however, one might as well use it for code in the kernel terminology as well, and then such could not be compiled by the standard kernel. So, why not add a NOT?

George B. Lyons
Jersey, City, NJ

EMPLOYMENT WANTED

Chairman of the FORTH Bit Slice Implementation Team (4th BIT) desires a junior programmer position working in a FORTH environment. (Also know COBOL & BASIC.)

Contact:
Martin Schaaf
202 Palasades Dr.
Daly City, CA 94015
(415) 992-4784 (eves.)

HELP WANTED

HELP 4TH BIT

With the implementation of a FORTH machine in AMD bit slice technology. If you're a hardware or microcode expert we can use your help. (This is a volunteer FORML project.)

Contact:
Martin Schaaf
Chairman, 4th Bit
202 Palasades Dr.
Daly City, CA 94015

MEETINGS

LA fig User's Group
October 1980

The LA group continued to experiment with format on its second meeting. It will continue to meet on the fourth Saturday each month at the Allstate Savings and Loan located at 8800 S. Sepulveda Blvd., 1/2 mile north of the LA airport.

The agenda this month called for a FIG meeting at 11, lunch at noon, and a FORML session at 1 patterned after our northern neighbors.

ARE YOU A — — — - - - - FIGGER?
YOU CAN BE!
RENEW TODAY!

At 11, a 20 minute random access was followed by an introduction by each of the 40 people present. The remaining half hour before lunch was evenly divided between a summary of the FORTH '79 document given by Jon Spencer, and a series of short announcements. These included a reminder about the Asilomar happening, a query about target compilers for the figFORTH environment, a suggestion that the LA and northern CA group exchange copies of notes or handouts from the meetings, a brief interchange of thoughts on program exchange leading to the idea of a uniform digital cassette standard, requests for assemblers and model corrections, and finally a parallel was drawn between the science fiction group's use of an "Amateur press association" as a potentially useful distribution channel.

From 1:15 to 4, Jon Spencer presented a FORML section which covered 3 topics:

1. Language processing.
2. Address binding and examples of a FORTH linker.
3. A continuation of his talk of last month on an algebraic expression evaulator for FORTH.

We all offer our thanks to Phillip Wasson who has organized the LA group. He is available at 213-649-1428 for details of the coming meeting. To get things rolling as far as program and information exchange, I volunteered as the LAFIG librarian. In 2 sessions, this has already expanded to writing a review for FORTH Dimensions and keeping track of spare copies of the handouts. I can be reached evenings from 7 to midnight at 213-390-3851.

;s Barry A. Cole

L.A. fig Meeting
November 1980

The November meeting was slightly smaller and less formal than the preceding meetings. After a short round of introductions, we were treated to a demo of a new set of FORTH system/application tools by the author, Louis Barnett of Decision Resources Corp. He has an Advanced Directory, File, and Screen Editor system which fits on top of fig-FORTH. I have looked at implementing a similar system in the past. He has thought out the tradeoffs of flexibility, speed, and keeping compatible with existing FORTH block formats. He allows the blocks to be interpreted in the traditional manner (by block #), as well as by file name and relative block number. He uses buffer pools and bitmaps to use all available disk space. It keeps a list of block numbers within a named file. Best of all, it allows editing, printing, and compiling by named file. I was sufficiently impressed to buy a copy on the spot.

After lunch, I presented an introduction to a tool I have been working on. It is used to build stack diagrams interactively for screens or colon definitions from the source screen coupled with symbolic element names entered from the console. I will write it up for a future issue of F.D.

;s Barry A. Cole

RENEW NOW!

RENEW TODAY!

LA MEETING

The next meeting of the "L.A. FORTH Users Group" will be

at: Allstate Savings & Loan
Community Room
8800 S. Sepulveda Blvd.
Los Angeles, CA
(1/2 mile north of LAX)

January 24, 1980 ("FORTH" Saturday)
11-12 AM General session
12- 1 PM Lunch break
1- 3 PM FORML Workshop

Info: Philip Wasson (213) 649-1428

FORML
October 1980

Henry Laxen opened with a discussion session on the problems of teaching FORTH. This produced a number of ideas ranging from subglossaries and reorganizations of glossaries, to comments on style and the categorization of tools. An anecdote by Kim Harris described a class of experienced FORTH programmers all FORTHing a traffic intersection problem only to be startled to discover that Charles Moore's solution used no IFs (the dictionary already is a link of IFs !)

Northern California
November 1980

FORTH-79 STANDARD: Bill Ragsdale summarized details of the just-published standard which had been worked out last year at Catalina Island. Handed out was a FORTH-79 Standard HANDY REFERENCE card and a two-page FORTH-79 Standard Required Word Set and requirements sheet with system errors and errors of usage specified. About vocabulary chaining, Bill mentioned the European approach--dynamic and oneway. In contrast, FORTH, INC. has a 4 level chain and the FORTH-79 Standard uses explicit chaining by vocabulary-name invocation.

Handouts included a FORTH machine proposal by Martin Schaaf, Ragsdale's CASE statement, a workshop announcement (for December) and Product Reviews of Laboratory Microsystems' Z-80 fig-FORTH and SBC-FORTH from Zendex Corp., by C.H. Ting. Introductions included:

- Sam Bassett is writing a text on FORTH For Beginners.
- Kim Harris' Humbolt State Univ. class will be held the week of 23-27 March.
- Ron Gremban offered a 4th programming job.
- FORTH will be mentioned in the next WHOLE EARTH CATALOG.
- Bill Ragsdale had been elected to the Board of Directors of FORTH, INC.
- Future fig meetings will be held underneath Penneys just East of Liberty House, Hayward.

;s Jay Melvin

FORML
November 1980

FORML - Klaus Schleisiek spoke about his FORTH implementation of an audio synthesizer which we heard on a cassette recording. The input device has a lightpen and output was by 64 speakers. Digital counters were organized in a linked list of registers comprising a table of sounds searched by NEXT. The structure of Klaus' program was depicted in discussion and on a half dozen xeroxed screens.

Northern California
October 1980

Bok Lee described STOIC, "A baroque elaboration of FORTH". This dialect differs from figFORTH by virtue of its third stack (Loopstack for I parameters) and its 4th stack which handles up to four vocabularies used, a compile buffer (which can be simulated by :: definitions in FORTH) and by its file system which is not screen-dependent but of indefinite length. The STOIC presentation was followed by a panel debate consisting of Kim Harris, Bob Fleming, Dave Bolton, Bill Ragsdale and B.W. Lee where it was unanimously decided "to each his own". General agreement was made about STOIC or FORTH's ability to simulate features of each other. The following differences seemed noteworthy:

- STOIC has some old style (FORTRAN?) mechanisms reflecting author Sack's incomprehension of some of author Moore's concepts.
- STOIC is conceptually not verbal, as is FORTH.
- STOIC is very well documented!
- STOIC is not supported by a group (like fig) and, consequently,
- STOIC is not portable.

Mr. Bok's handouts included a (sample) DUMP program, NORTHSTAR and CP/M memory maps for STOIC and a decompiler. Other meeting handouts included a structured (FIND) by Mike Perry (which appears to be 8080 coded), a 6502 assembler with heavy commenting by Tom Zimmer as well as Zimmer's ad for tiny PASCAL, ROM and disk based OSI FORTH and Asilomar FORML details. C.H. Ting introduced his just published <u>FORTH SYSTEMS GUIDE</u>, which is enlightening. Sam Daniel volunteered to take on my scribeship abandoned due to marriage and relocation in L.A.

;s Jay Melvin

FORTH COURSE

PEOPLE, COMPUTERS, AND
FORTH PROGRAMMING

DATE

March 23-27, 1981

COURSE

The course is an intensive five day program on the use of FORTH. Topics are to incluse usage, extension and internals of the FORTH language, compiler, assembler, virtual machine, multitasking operating system, mass storage, virtual memory manager, and file system. Computers will be used for demonstrations and class exercises. Due to class size limitations only twenty participants will be permitted. Please register as soon as possible but no later than March 1, 1981. The cost will be $100, or $140 with 3 units of credit. The manual "Using FORTH" will be available for an additional $25.

Send payment to:

Barbara Yanosko
Office of Continuing Education
Humboldt State University
Arcata, CA 95521

LOCATION

Humboldt State is located in Arcata, California, six miles north of Eureka and about 300 miles north of San Francisco. Arcata has bus and plane service from San Francisco and Portland. Motels are available for lodging. Transportation will be available from the local "Motel 6". Other motels are within walking distance. For reservations, contact:

Motel 6
4755 Valley West Blvd.
Arcata, CA 95501

INFORMATION

For other information contact:

Professor Ronald Zammit
Physics Department
Humboldt State University
Arcata, CA 95521

(707)826-3275
(707)826-3276

FIG CONVENTION

The second annual FIG Convention was a big success with 250 FORTH users, dealers, and enthusiasts attending a full day of sessions on FORTH and FORTH-related subjects. The Villa Hotel in San Mateo, CA provided the setting this year.

In the annual report, Bill Ragsdale mentioned some of the milestones passes by FIG in 1980:

1. Total membership is now 2,044. About 1200 new members joined this year, primarily due to the BYTE issue devoted to the FORTH language.

2. Roy Martens was hired this year as the full-time publisher of FORTH DIMENSIONS, and is also taking over responsibility for all mail-order and telephone inquiries.

3. The first college-level course in FORTH was taught by Kim Harris in 1980. Another course, to be offered in 1981, will give college credit for completion.

4. The FORTH-79 Standard was approved just prior to the convention, and copies are available through FIG mail order.

5. Regional groups are springing up all over the U.S. New groups are now meeting in Los Angeles, Boston, Dallas, San Diego, San Francisco, and approximately 20 other cities across the country.

Following a panel session on the FORML conference at Asilomar, Charles Moore of FORTH, Inc., closed the morning session with a reminder that it is the very flexibility and versatility of FORTH which will cause more problems as more people become acquainted with it. In particular, we must be able to demonstrate to large mainframe users that FORTH is also applicable in their environment.

The afternoon session was highlighted by two very interesting presentations. The first was on software marketing, pointing out very clearly the differences in professional and amateur approaches to selling of software. The second presentation was by Dr. Hans Nieuwenhuizen, of the University of Utretch in Holland, regarding the implementation of High Level Languages in FORTH. Dr. Nieuwenhuizen reported running BASIC, PASCAL, and LISP systems, written entirely in FORTH, at the University of Utretch. (Please do not write Dr. Nieuwenhuizen concerning availability of this software. When it is ready for distribution, an announcement will be made through FORTH DIMENSIONS.)

The formal part of the convention concluded with presentations from some of the many vendors of FORTH systems and software.

After a short interlude for informal discussion and attitude adjustment, Mr. Allen Taylor, author of the Taylor Report in ComputerWorld, was the guest speaker at the now-traditional evening banquet.

;s S. Daniel

MORE LETTERS

Since I seem to be the first OSI user to have the FIG model installed and fully operational, I thought that you might add my company name to your list of vendors. I have been extremely faithful to the model, changing only the I/O and -DISC. Everything works just fine, and by that I mean a lot better than OSI's standard system software. I did find a miscalculated branch (forward instead of backward) and the address of ;S was left off of the end of UPDATE (with lethal result) in case you are interested. Unfortunately, I couldn't use the ROM monitor for MON since it blows out the OSI DOS. Instead MON jumps to the DOS command interpreter, which is more useful than the OSI ROM monitor, anyway.

I feel that I am in a position to fully support the system, since I know OSI's hardware and DOS inside-out, and also it appears that I may have their (OSI's) cooperation and even mention in future advertisements.

I have enclosed a press release which describes system requirements, ordering information, and price.

Guy T. Grotke
San Diego, CA

We are soliciting comments, suggestions and bug reports concerning the fig-FORTH 8080 source listing. Work on converting this to the 1979 Standard will begin in early Februry, 1981, so please make submissions as soon as possible to:

8080 Renovation Project
c/o FORTH Interest Group
P.O. Box 1105
San Carlos, CA 94070

Terry Holmes

Dear FIG (Whoever you are),

Just a little note to let you know that I received all the FIG material that I ordered. I would like to know if the 8080 listing is available on IBM formatted single density 8" diskettes and if the fig-FORTH model listed in the Installation Manual, i.e., Screen Nos. 3-8, 12-80, 87-97, is also available on an IBM format 8" single density diskette? I don't relish having to type in all that material, to get fig-FORTH up and running.

I have taken the liberty to spread the word about fig-FORTH in my computer club and have attached copies of two of our newsletters, in which reference to it has been made, see VCC NL Issue 109- bottom p. 3 and Issue 112- middle p.2.

S. Lieberman
Valley Computer Club
P.O. Box 6545
Burbank, CA 91510

(An 8080 figFORTH system on 8" diskette for CP/M systems is available from Forthright Enterprises - P.O. Box 50911, Palo Alto, CA 94303 -- Ed.)

Here is a program you are welcome to publish in FORTH Dimensions.

Lyall Morrill
San Francisco, CA

```
SCR # 222
( ENIUQ A 'Self-Rep' after Douglas R. Hofstadter 17 AUG 80 )

     FORTH DEFINITIONS  DECIMAL

( Self-reproducing source code in two lines of fig-FORTH.  When
loaded, types a copy of itself.  Note the 128th character. )


: ENIUQ CR 34 WORD HERE COUNT 2DUP TYPE CR TYPE 34 EMIT : ENIUQ
: ENIUQ CR 34 WORD HERE COUNT 2DUP TYPE CR TYPE 34 EMIT : ENIUQ"


( See "Godel, Escher, Bach:  An Eternal Golden Braid,"
by Douglas R. Hofstadter, Basic Books, Inc., 1979, page 498. )

     ;S   LDM
```

Just a line to let you know of a couple of FORTH activities at this end of the country. Here at Temple U we have a lab equipped with 25 AIM systems. Microprocessor Systems is a 56-hour lecture / 28-hour hands-on course of which about 12/6 hours are allotted to AIM Assembler.

I am now testing both Rehnke's V 1.0 FORTH cassette and Rockwell's V 1.3 FORTH ROM chips. I expect to teach one or the other in place of the AIM Assembler this term.

On March 21st the IEEE UPDATE Committee is running an all-day tutorial on FORTH. At that time I hope to demonstrate FORTH transportability between, say, AIM and PET or Apple. I wonder whether anything has been published on this sort of demonstration.

Karl V. Amatneek
Director of Education
Committee for Professional
UPDATE
Wyndmoor, PA

(No, but if you'd like to write one... Ed.)

I get a great deal of your mail. I work for GTE LENKURT, 1105 Old County Road, San Carlos. Those idiots in the post office can't distinguish that from P.O. Box 1105 and our names are not that dissimilar, I guess.

Please get another box number.

M. Mohler
San Carlos, CA

(Guess we're TOO popular -- Ed.)

The video editor presented as an example of CASE use by Major Robert Selzer in FORTH DIMENSIONS v. II/3, p. 83 is super.

Enclosed is a direct extension of Major Selzer's work to edit ASCII files over several consecutive screens. It is used in the form:

```
n1 n2 FEDIT
```

where n1 is the first screen in the file and n2 is the last.

FEDIT contains all the commands of Major Selzer's VEDIT and works in the same manner. ESC exits the editor and the cursor position is controlled by the single keystrokes LEFT, RIGHT, UP, DOWN AND RETURN. When the top or bottom boundary of the display is reached a new display of either the next or the previous 24 lines in the file is presented for editing.

The added commands are RUB which deletes characters and two double keystroke commands HOME and TAB.

HOME followed by DOWN or UP produces a display of the next or previous 24 lines respectively independent of the position of the cursor. Two successive strokes of HOME produce a new display with the line containing the cursor in the old display at the center of the new display. These commands provide rather rapid traversal of a file and positioning of the file on the display.

At the end of a file, additional numbered but blank lines may be displayed. Text written into this area will not be put into the buffer. Similarly if the first line of the file ends up in the middle of a display, the area above the first line is protected.

The TAB key is used to erase, delete and replace lines from PAD. TAB followed by E erases the line containing the cursor. TAB followed by D erases the line and holds the line in the text output buffer PAD. The cursor may then be moved to any position in the file, including other screens, and the contents of PAD may be put on the new line by the keystrokes TAB then P. TAB followed by H places the cursor's line in PAD without deleting the line. These commands use the fig-FORTH line editor definitions E, D, R (REPL in the listing) and H.

Major Selzer's definition of CASE does not work in fig-FORTH with its compiler security features. An appropriate definition of his CASE word for fig-FORTH is shown on line 10. The word OFF on line 68 controls a switch in my EMIT to stop output to my printer. All other words should be standard fig-FORTH. The terminal dependent cursor position sequence used by Selzer for his ADM-3A terminal (YXCUR, line 3) also works on my SOROC IQ 120 terminal.

I have found FEDIT to be a convenient editing tool which I use along with the fig-FORTH editor. Eventually, I suppose, my entire fig-FORTH editor will find its way into FEDIT. I hope your readers will also find it convenient. I also hope FEDIT lays to rest some of the recent criticism of FORTH (in BYTE) concerning its rudimentary editing facilities. My thanks to all of you in FIG for your efforts in promotion FORTH.

Edgar H. Fey
LaGrange, IL

```
SCR # 64
 0 ( ASCII FILE EDITOR  SCR 64 TO 68     E H FEY Corr 11/2/80)
 1   0 VARIABLE CUR  0 VARIABLE COFF  0 VARIABLE N2  0 VARIABLE N1
 2
 3 : YXCUR ( x y ... ) 27 EMIT 61 32 + EMIT 32 + EMIT ;
 4 : .CUR  ( ... ) ( Print cur ) CUR @ 64 /MOD SWAP 4 + SWAP YXCUR ;
 5 : !CUR  (n ... ) ( Store n in cur ) 0 MAX 1535 MIN CUR ! ;
 6 : +CUR  (n ... ) ( Add n to cur ) CUR @ + !CUR ;
 7 : +.CUR (n ... ) (Add n & print cur ) +CUR .CUR ;
 8 : HOM   ( ... ) ( Reset cur ) 0 CUR ! ;
 9 : (CASE) OVER = IF DROP 1 ELSE 0 THEN ;
10 : CASE COMPILE (CASE) [compile] IF ; IMMEDIATE
11
12 : .LB ( l blk ... ) ( Print line l of block blk if blk in file )
13   DUP N1 @ < OVER N2 @ > OR ( ...l blk blk>n2O Rblk<n1 )
14   IF ( Not in file ) DROP DROP 68 0 DO 32 EMIT LOOP
15   ELSE ( In file ) DUP >R (LINE) TYPE R> 124 EMIT . THEN ;      -->

SCR # 65
16
17 : .NL (n l b ... ) ( Print n lines from line l relative to lin )
18   ( 0 of block b ) OVER DUP >R 16 /MOD   ( ...n l b reml l/16 )
19   R> 0< IF ( l<0 ) ROT + -1 + SCR ! 16 + ( ...n l reml+16 )
20   ELSE ( l>=0 ) ROT + SCR !              ( ...n l reml )
21   THEN  SWAP ROT OVER >R + >R            ( ...rel l+n l )
22   DO ( n lines )                         ( ...rel )
23   CR I 3 .R SPACE DUP SCR @ .LB 1+ /MOD SCR +! LOOP DROP ;
24
25 : ACUR ( ... acur ) ( Abs cursor addr in file ) CUR @ COFF @ + ;
26 : (+LIN) ( n ... cur ) ( Computer CR+nLF ) CUR @ 64 / + 64 * ;
27 : ACMX ( ...acurmax ) N2 @ N1 @ - 1+ B/BUF * ;
28
29 : .HOME ( n ... ) ( New display, line of old cursor at line n )
30   ACUR 64 / OVER - 24 OVER N1 @ .NL
31   64 *    COFF !  HOM 64 * +.CUR ;                              -->

SCR # 66
32 : .TOP ( n... ) COFF @ 0= IF DROP ELSE +CUR 23 .HOME THEN ;
33
34 : +.ACUR ( n ... ) ( ADD n to abs cursor.  Display cursor )
35   DUP CUR @ + DUP 0<                     ( ... n n+cur f )
36   IF ( OFF top ) DROP .TOP               ( ... )
37   ELSE ( Not off top ) 1535 >            ( ...n n+cur>1535 )
38      IF ( Off bottom ) +CUR ACUR ACMX < IF 0 .HOME THEN
39      ELSE ( In display ) +.CUR  THEN THEN ;
40
41 : +LIN ( n ... ) ( Add n lines to cur, CR+nLF ) (+LIN) !CUR ;
42 : +ALIN ( n ... ) ( Add n lines to curs ) (+LIN) CUR @ - +.ACUR ;
43 : BOX ( ...f ) ( True if in file ) ACMX ACUR > ACUR -1 > AND ;
44 : !ABLK ( c... ) ( Store ascii char in buffer ) BOX IF ( In file )
45   ACUR B/BUF /MOD N1 @ + BLOCK + C! UPDATE 1  +.ACUR THEN ;
46 : RUB 32 DUP EMIT !ABLK -2 +.ACUR ; ( Del char, retreat curs )
47 : ALIN ( ...l ) ( Get cursors scr and line                     -->

SCR # 67
48    ACUR B/BUF /MOD N1 @ + SCR ! 64 / ;
49 : E ( ... ) ( Display blank line )
50   0 + ALIN 64 0 DO 32 EMIT LOOP 0 +ALIN ;
51 : .P ( ... ) ( Replace line from PAD at cur line & display )
52   ALIN REPL 0 +ALIN ALIN SCR @ .LB 0  +.ACUR ;
53
54 : TAB2 ( ... ) ( 2nd key stroke for choice of TAB )
55   KEY 69 CASE ALIN E .E         ELSE ( E=Erase curs line )
56       72 CASE ALIN H            ELSE ( H=Hold curs line at PAD )
57       80 CASE  .P               ELSE ( P=Replace line from PAD )
58       68 CASE ALIN H ALIN E .E ELSE ( D=Del line, hole in PAD )
59   DROP 0 +ALIN THEN THEN THEN THEN     ; ( Default CR no LF )
60
61 : HOME2 ( ... ) (2nd key stroke choice of HOME ) KEY
62    10 CASE 1535 +CUR 0 .HOME  ELSE ( Down=scroll next )
63   11 CASE -1535 +CUR 23 .HOME ELSE ( Up=scroll prev )           -->

SCR # 68
64    30 CASE       12 .HOME      ELSE ( Home=center cursor )
65   DROP 0 +ALIN THEN THEN THEN ;    ( Default= CR no LF )
66
67 : FEDIT ( n1 n2... ) ( Edit file in blocks n1 to n2 incl. )
68  N2 ! N1 ! OFF 0 COFF ! HOM 0 .HOME BEGIN
69  KEY 27 CASE    0 23 YXCUR QUIT     ELSE ( ESC )
70        8 CASE      -1  +.ACUR       ELSE ( LEFT )
71       10 CASE      64  +.ACUR       ELSE ( DOWN )
72       11 CASE     -64  +.ACUR       ELSE ( UP )
73       12 CASE       1  +.ACUR       ELSE ( RIGHT )
74       13 CASE       1  +ALIN        ELSE ( RETURN )
75      127 CASE        RUB            ELSE ( RUB )
76        9 CASE     7 EMIT TAB2       ELSE ( TAB, Next key picks )
77       30 CASE     7 EMIT HOME2      ELSE ( HOME, Next key picks )
78   DUP EMIT !ABLK THEN THEN THEN THEN THEN THEN THEN THEN THEN
79   AGAIN ;                                                       ;S
```

THE FORTH SOURCE

A wide variety of FORTH printed material, both public domain and copyrighted, is available. Send for list:

Mountain View Press
PO Box 4656
Mt. View, CA 94040

NEW PRODUCTS

6800 & 6809 FORTH

† FORTH FORTH System	$100
† FORTH+ plus Assembler, CRT Editor	$250
firmFORTH produces compacted ROMmable code	$350

Kenyon Microsystems
3350 Walnut Bend
Houston, Texas 77042
Phone (713)978-6933

CRT EDITOR AND FILE MANAGEMENT SYSTEM

The Decision Resources File Management System (FMS-4) for the FORTH language has extensive vocabulary for creating, maintaining and accessing name files.

Disk space is dynamically allocated and deallocated so there is never any need to reorganize a disk. From the user viewpoint, access is to logical records; FMS-4 performs the mapping to physical screens.

Files may be referenced by name without concern for the physical location of the file on disk. FMS-4 supports sequential and direct access while preserving FORTH's facilities for addressing screens by number.

FMS-4 maintains a file directory of up to 47 entries. Each file may consist of from one to 246 records (1024 bytes per screen) in a single volume (single density diskette). It is also possible to extend FMS to control multiple volume files and to support larger directories.

In addition to an extensive command set, there are many lower level primitives which may be combined to define a virtually unlimited set of commands.

Computer system hardware should include:

One or more 8" IBM compatible floppy disk drives

Enough memory to support 6K bytes (on an 8 bit processor) for FMS-4 in addition to the FORTH nucleus and any other concurrently resident applications.

An 8080/8085 or Z80 cpu.

A CRT or printing terminal which supports upper and lower case.

System software should include:

fig-FORTH compatible nucleus or equivalent.

An assembler for the target cpu. DRC can supply an 8080 assembler at additional cost.

FMS-4 source code is delivered ready to run (on compatible systems) on a single density 8" soft sectored diskette (IBM 3740).

A complete user manual describing all facets of FMS-4 operation is provided. The manual includes an extensive glossary which defines and documents the usage of each word in the FMS vocabulary.

Wordsmith is a CRT screen editor which is integrated with Decision Resources' File Management System - FMS-4. The combination is an especially powerful file oriented editor which combines the extensive disk space management facilities of FMS-4 with the flexibility and immediacy of on-screen editing.

The full record being edited is continuously displayed on the CRT and all changes are immediately visible. There are 41 editing commands including: multidirectional cursor movement, record to record scrolling, record insert and delete, string search and replace, text block movement and many more.

The FMS-4 and Wordsmith Packages

Wordsmith and FMS-4 source code is delivered ready to run (on compatible systems) on a single density 8" soft sectored diskette (IBM 3740).

Complete user manuals for each system are provided.

Pricing

Single noncommercial user license:	
FMS-4	$50
Wordsmith (with FMS-4)	$95
Manual only:	
Wordsmith	$15
FMS-4	$15
Both	$25

(credited toward purchase of full package)

California residents add 6% sales tax
Shipping and handling: $2.50

Commercial Purchasers should contact Decision Resources.

Decision Resources Corporation
28203 Ridgefern Court
Rancho Palos Verdes, CA 90274
(213) 377-3533.

TRS-80 DISKETTES

Advanced Technology Corp. of Knoxville, TN, is presently distributing its fig-STANDARD FORTH version (TFORTH) customized for the Radio Shack TRS-80. Included in this package are: assemblers, 'TRACE' function for generating minimum system /CMD files, POINT, SET, CLS commands for graphics use, Floating point package, I/O package (LPT Output) and variable number base to base 32.

The language is supplied on either 80 or 40 track 5-1/4" diskette for $129.95 and the manual is also included.

This product may be purchased from:

Sirius Systems
7528 Oak Ridge Highway
Knoxville, TN 37921

or

QC Microsystems
P.O. Box 401326
Garland, TX 75040

or directly from us,

Advanced Technology Corp.
1617 Euclid Avenue
Knoxville, TN 37924
(615) 525-1632

PRODUCT REVIEWS

by
C.H. Ting

Z-80 fig-FORTH by Ray Duncan of Laboratory Microsystems, 4147 Beethoven St., Los Angeles, CA 90066 (213) 390-9292.

Two 8" single density diskettes, $25.00.

The first disc is a CP/M disc containing Z-80 assembly source codes, hex object codes, user instructions, fig-FORTH Installation Manual, and fig-FORTH Glossary. The second disk is in FORTH block format containing system configurations, a line editor, a poem 'The Theory That Jack Built' by F. Winsor, Eight Queens Problem by J. Levan, Towers of Hanoi by P. Midnight, Breakforth by A. Schaeffer, and some utilities.

I do not have a system that can run the Z-80 codes. However, the source codes seem to be carefully done and follow faithfully the fig-FORTH 8080 model. Lots of typing was put in to have the entire Installation Manual and Glossary entered on disc. The games were published in FORTH Dimensions. The amount of information offered at this price is unbelievable. I just wish that I had a machine that could run it.

SBC-FORTH from Zendex Corp., 6398 Dougherty Rd., Dublin, CA 94566 (415) 829-1284.

Four 2716 EPROM's to run in an SBC-80/20 board with SBC-201 single density disk. $450.00.

I had the PROM's installed in a System 80/204. It ran only after I jumpered the CTS/ and RTS/ pins of the 8251 serial I/O chip. Obviously the chip uses some interrupt scheme to drive the terminal. I was not able to get the detailed information on how the interrupts were supposed to go from Zendex. I do not have a disc drive in the system to test out the disc interface. Other things ran satisfactorily. I was able to talk to the parallel I/O ports using the assembler.

This type of ROM based FORTH machine can be very powerful for programmable controllers and low cost development systems if some non-volatile memories like core or battery-backed CMOS were added.

A very nice thing they did in the manual was to include the code or colon definitions in the Glossary, making it infinitely more useful as a reference.

NEW PRODUCTS

APPLE figFORTH

Including an Assembler, Screen Editor, Source Code and associated compiler, with some documentation on disk. No other documentation, support or instruction. Source listing will be available from fig in mid-81. Apple format disk - $30.00.

George Lyons, 280 Henderson St., Jersey City, NJ 07302.

CROMEMCO DISKETTE

A fig-FORTH 5-1/2" disk with Z80 assembler for Cromemco machines. $42.00

Nautilus Systems
PO Box 1098
Santa Cruz, CA 95061

NEW PRODUCTS

"Systems Guide to fig-FORTH"

Author: C.H. Ting
156 14th Ave.
San Mateo, CA 94402

132 pages, $20.00

This book is meant to be a bridge between "Using FORTH" and the "fig-FORTH Installation Manual", and to serve as a road map to the latter. It might also be used as a collection of programming examples for those studying "Using FORTH".

In it, I have tried to arrange the fig-FORTH source codes into logical groups: Text Interpreter, Address Interpreter, Error Handler, Terminal I/O, Numeric Conversions, Dictionary, Virtual Memory, Defining Words, Control Structures, and Editor. Extensive comments are thrown in between source codes at the risk of offending the reader's intelligence. Occasionally flow charts (horror of horrors!) are used to give graphic illustrations to some complicated words or procedures.

There is a very wide gap between the front page and the back page of the FORTH Handy Reference Card. It is relatively easy to manipulate the stacks and to write colon definitions to solve programming problems. The concepts behind words of system functions, like INTERPRET, [,] , COMPILE, VOCABULARY, DEFINITIONS are very difficult to comprehend, not to mention <BUILDS and DOES> . One cannot understand the FORTH system and how it does all these wonderful things by reading the source codes or by searching the glossary. These documents are vehicles to define the FORTH system, not to promote understanding of them.

OSI DISC

Tiny PASCAL written in fig-FORTH. Machine Readable for OSI-C2-8P. Single or Dual Floppy System 8" disc.
Cost: $60.00
This includes fig-FORTH with fig editor and assembler for FREE!

OSI C2 or C3 fig-FORTH on 8" disc.
Cost: $45.00
Includes assembler and fig editor

Tom Zimmer
292 Falcato Dr.
Milpitas, CA 95035
(408) 245-7522 ext. 3161 or
(408) 263-8859.

---

tinyPASCAL

Printed listing of tinyPASCAL in fig-FORTH.

$10.00 US/Canada, $14.00 Overseas. Check (US bank), VISA or Master Charge.

Mountain View Press
PO Box 4656
Mt. View, CA 94040

FORTH Version 1.7

Cap'n Software FORTH Ver. 1.7 for Apple II (TM) or Apple II+ computers is the FORTH Interest Group (FIG) language, plus extensive program development tools and special Apple options. $175.00.

Cap'n Software
P.O. Box 575
San Francisco, CA 94101.

SEPARATED HEADS

Klaus Schleisiek

Memory in RAM-based systems can be used more efficiently by means of a "Symbol Dictionary Area," which allows words and/or name and link fields which are needed only at compile time to be thrown away after compilation. Incremental use of these techniques will result in more efficient memory usage and will also encourage the use of more and shorter definitions because there is no longer the need to pay the penalty of taking up memory space with numerous name and link fields.

In the course of a two-year project I developed some tools which allow a significant compression of code in RAM-based systems. I also feel these methods will have a significant impact on programming style, particularly because they will encourage the use of more and shorter definitions. The following is an explanation of these various functions in a somewhat historical order.

My programming task was to develop a lightpen-operated sound system, which would allow control of a number of small sound synthesizers by pointing a lightpen to various dots, light potentiometers, and the like on a video display. There was to be no keyboard intervention. A "dot" was put together by compiling a word which associated the following information:

A) The shape of the dot itself as an address of some programmable character.

B) The dot's location on the screen as an address relative to the upper left hand corner of the screen.

C) As an option, either a text string or a string of programmable graphics characters to be displayed above, below, or to either side of the dot.

Thus, every "dot" served a double purpose. On the one hand, it described a portion of the display itself which had to flash on the screen. Secondly, it supplied the key to a large keyed CASE statement which associated the dot with the function to be performed when the lightpen was pointed to it. In other words, the definitions of the dots themselves were only needed at compile time.

The dot definitions were used to create a densely packed "image" definition to flash the picture on the screen, while the addresses of the dot locations were used as keys in the CASE statement. So, to be memory efficient, I wanted to set up some mechanism which would allow the presence of "symbols" at compile time that could then be thrown away after compilation to free memory. By "symbol" I mean any legal FORTH definition that is only needed at compile time. This led to the idea of dividing the dictionary into "main dictionary" and " symbol dictionary."

Figure 1 shows the arrangement of this scheme based on the 6502's unique memory mapping.

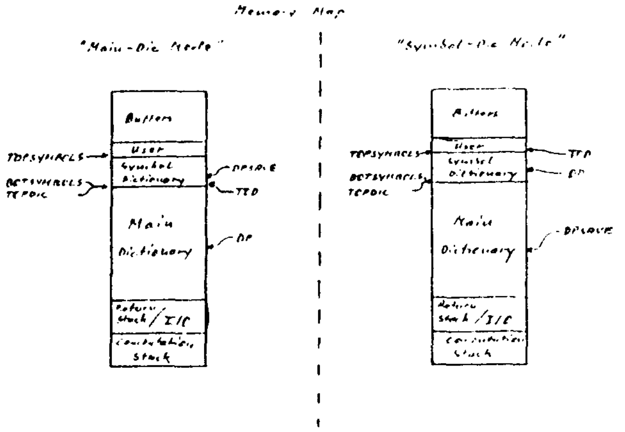


Figure 1

I soon realized that most of the words defined in my programs would never be used again after compilation and started thinking about putting the name and linkfield (head) of a definition into the symbol dictionary, and compiling the code field and parameter field only into the main dictionary. I wanted to do it in a fashion similar to SYMBOL DIC and MAIN DIC .

This would mean switching back and forth between one state which compiles the heads into the symbol dictionary and another state which compiles the heads as usual. This switching is done by the variable HEADFLG (SCR #23) which is respectively set and reset by DROP-HEADS and COMPILE-HEADS (SCR #23). The state of HEADFLG in turn changes the behavior of CREATE (SCR #24).

One complication is that the use of HEADFLG interferes with the symbol dictionary mechanism: If you are compiling into the main dictionary, you want the dropped heads to be compiled into the symbol dictionary, but if you are compiling into the symbol dictionary anyway, you want the heads to go there too.

In other words, in the first case the body of a definition would be separated from the head, while in the second case, body and head would not be separated. This requires the redefinition of CREATE (SCR #24) and the use of three values for HEADFLG. The first two states are set explicitly by COMPILE-HEADS and DROP-HEADS, but the third state is recognized and handled by CREATE .

When a word is compiled, its name field and link field are compiled into the symbol dictionary and the word is made immediate and (CFA) is compiled as its code field, followed by the address of the next memory location in the main dictionary. The remainder of the current definition (the body) will then be compiled into the main dictionary. When references are made to the word, its CFA is contained in the memory location next to the code field address of (CFA) .

The function of (CFA) (SCR #23) is either to compile the execution address of code into the dictionary (when the word is subsequently used in a definition), or to execute the definition, depending on STATE, before forgetting the symbols. The implementation described here deals with the 6502 and has to deal with the idiosyncrasy that no CFA may be located at XXFF, which in turn makes the definition of (CFA) and CREATE somewhat mysterious!

FORGET-SYMBOLS (SCR #22) is the word which "rolls" through every dictionary and "unlinks" every definition which was placed in the symbol dictionary, thereby freening it (Figure 2). It is somewhat slow and it is assumed that no symbol exists below FENCE @ . Before forgetting anything in the main dictionary, however, you <u>must</u> FORGET-SYMBOLS . Otherwise links may be broken and the interpreter won't work anymore.

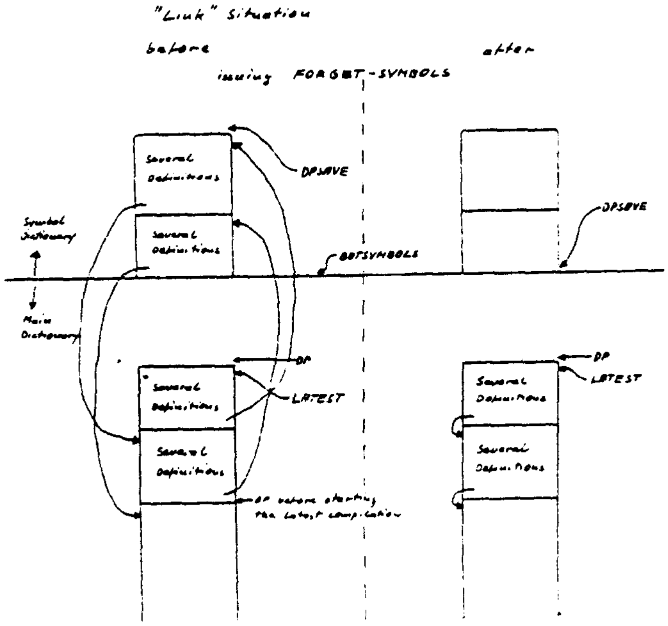


Figure 2

The next step was to make the defining words work as well in the DROP-HEADS mode, which meant that (;CODE) had to be redefined (SCR #25). It now uses the subdefinition (;COD) and depending on the state of HEADFLG, determines the location of the code field to be rewritten and rewrites it.

A problem might arise in the rare case where a definition whose head is to be thrown away is supposed to be immediate by itself. The "solution" to this problem was to simply declare such a case illegal. There is a reason, however. The only situation where one might want an immediate definition to be placed in the main dictionary would be in coincidence with [COMPILE] within some definition. Otherwise, one would want to compile it entirely into the symbol dictionary anyway. Such a case is so rare that it did not seem worth the effort to redefine IMMEDIATE and [COMPILE] . To use a word whose head has been compiled in to the symbol dictionary immediately within a definition, one has to use .... [XXXX] .... !

Finally, I observed that I was generating <BUILDS ... DOES> and ;CODE constructs with big compile time definitions, which do nothing but take memory space at execution time. But dropping the heads of <BUILDS ... DOES> means that the compile time parts of these definitions won't ever be used at compile time either. Thus, if the heads of <BUILDS ... DOES> are dropped, everything prior to DOES> may be dropped as well. It will, however, be necessary to redefine DOES> and ;CODE to do this (SCR #26 and SCR #27).

At compile time, the situation of a <BUILDS ... DOES> construct is as follows: While in the DROP-HEADS state, the name has been put into the symbol dictionary and subsequently <BUILDS ... has been compiled into the main dictionary. When we come to DOES> everything which had been compiled into the main dictionary, including the code field, must be moved into the symbol dictionary.

This is done by MOVE-DEF? (SCR #25), which is used in DOES> and ;CODE . Depending on the state of HEADFLG , MOVE-DEF? either compiles (;CODE) or moves the previous definition into the symbol dictionary and compiles ((;CODE)) . ((;CODE)) has to be one step more indirect than (;CODE) and resembles the function of (CFA) in ordinary definitions.

A final note: The definitions for GOTO and LABEL, which allow multiple forward references (e.g., several GOTO's) may precede as well as follow "their" label. Even though I implemented this because it seemed more convenient than restructuring, there is some question as to its true value because it takes 318 bytes!

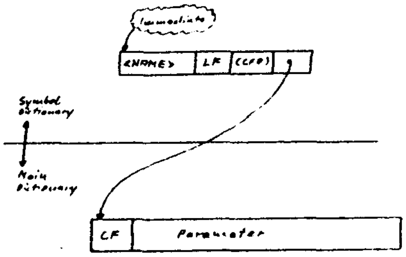


Figure 3

GLOSSARY

```
TOPDIC
        A CONSTANT THAT LEAVES THE NEXT BUT LAST ADDRESS
        TO BE USED AS MAIN DICTIONARY ON THE STACK.
BOTSYMBOLS
        A CONSTANT THAT LEAVES THE FIRST ADDRESS TO BE USED
        AS THE SYMBOL DICTIONARY ON THE STACK.
TOPSYMBOLS
        A CONSTANT THAT LEAVES THE NEXT BUT LAST ADDRESS
        TO BE USED AS SYMBOL DICTIONARY ON THE STACK.
DPSAVE
        A VARIABLE THAT CONTAINS THE DICTIONARY POINTER OF THE
        CURRENTLY INACTIVE DICTIONARY PARTITION.
TOD     ( TOP-OF-DICTIONARY )
        A VARIABLE THAT CONTAINS THE CURRENT NEXT BUT LAST
        MEMORY LOCATION TO BE USED FOR COMPILING DEFINITIONS.
MAIN-DIC
        RESETS DP TO POINT TO THE NEXT FREE MEMORY-LOCATION
        IN THE MAIN DICTIONARY.
        I.E. THE FOLLOWING DEFINITIONS ARE PERMANENTLY
        COMPILED INTO THE MAIN DICTIONARY.
        IF DP WAS ALREADY POINTING INTO THE MAIN DICTIONARY,
        IT DOESN'T DO ANYTHING.
        COUNTERPART: "SYMBOL-DIC"
```

```
SYMBOL-DIC
        SETS DP TO POINT TO THE NEXT FREE MEMORY LOCATION
        IN THE SYMBOL DICTIONARY.
        I.E. THE FOLLOWING DEFINITIONS WILL BE COMPILED INTO
        THE SYMBOL DICTIONARY AND MAY BE FORGOTTEN USING
        "FORGET-SYMBOLS" WITHOUT AFFECTING THE MAIN DICTIONARY.
        IF DP WAS ALREADY POINTING INTO THE SYMBOL DICTIONARY,
        IT DOESN'T DO ANYTHING.
        COUNTERPART: "MAIN-DIC"

FORGET-SYMBOLS
        IS USED FOR UNLINKING THOSE DEFINITIONS WHICH HAD
        BEEN COMPILED INTO THE SYMBOL DICTIONARY FROM
        THE MAIN DICTIONARY DEFINITIONS.
        RESETS THE SYMBOL DICTIONARY POINTER TO "BOTSYMBOLS" .
     WARNING: IF ANYTHING HAS BEEN COMPILED INTO THE
        SYMBOL DICTIONARY, YOU HAVE TO "FORGET-SYMBOLS"
        BEFORE FORGETTING ANYTHING IN THE MAIN DICTIONARY.
HEADFLG
        A VARIABLE THAT CONTAINS THE "HEAD-STATE" I.E.
        HEADFLG = 0  -> COMPILE-HEADS HAD BEEN ISSUED
        HEADFLG = 1  -> DROP-HEADS AND MAIN-DIC HAD BEEN ISSUED
        HEADFLG = 2  -> DROP-HEADS AND SYMBOL-DIC HAD BEEN
                        ISSUED.

COMPILE-HEADS                                               E
        COMPILE THE HEADS ( NAME & LINKFIELD ) OF THE
        FOLLOWING DEFINITIONS INTO THE MAIN DICTIONARY.
        COUNTERPART: DROP-HEADS

DROP-HEADS                                                  E
        THE HEADS ( NAME & LINKFIELD ) OF THE FOLLOWING
        DEFINITIONS WILL BE COMPILED INTO THE SYMBOL DICTIONARY,
        THE BODY (  CODE- & PARAMETERFIELD ) INTO THE MAIN
        DICTIONARY. FURTHERMORE, THE COMPILE TIME PARTS
        OF DEFINITIONS IN TERMS OF <BUILDS ... DOES> AND
        ... ;CODE WILL BE COMPILED INTO THE SYMBOL DICTIONARY
        TOO. ( I.E. EVERYTHING PRECEEDING ... DOES> OR
        ... ;CODE RESPECTIVELY WITHIN THE CURRENT DEFINITION )
        THE PARTS WHICH ARE LOCATED IN THE SYMBOL DICTIONARY
        MAY BE FORGOTTEN BY ISSUING "FORGET-SYMBOLS" WHICH
        EFFECTIVELY DISCARDS THE HEADS / COMPILE TIME CODE.
        BEFORE ISSUING "FORGET-SYMBOLS" THESE WORDS MAY BE USED
        IN THEIR USUAL MANNER FOR EITHER COMPILATION INTO
        HIGHER LEVEL DEFINITIONS OR EXECUTION.
     WARNING: DROP HEADS MAY NOT BE USED FOR IMMEDIATE
        DEFINITIONS. NO ERROR CHECKING IS PERFORMED !
```

```
SCR # 20
( SYMBOLDICTIONARY            KS  10-5-80 )
FORTH DEFINITIONS HEX

3800 CONSTANT TOPSYMBOLS
3000 CONSTANT BOTSYMBOLS
3000 CONSTANT TOPDIC

3000 VARIABLE TOD
3000 VARIABLE DPSAVE

: SWITCH-DIC
   HERE DPSAVE DUP @ DP ! ! ;

: ?MAIN-DIC  (  ---  F-1  )
   HERE BOTSYMBOLS U< TOPSYMBOLS HERE U< OR ;
-->
```

```
SCR # 21
( SYMBOLDICTIONARY            KS  10-5-80 )

: MAIN-DIC
   TOPDIC TOD ! ?MAIN-DIC 0= IF SWITCH-DIC THEN ;

: SYMBOL-DIC
   TOPSYMBOLS TOD ! ?MAIN-DIC IF SWITCH-DIC THEN ;

: ?SYMBOL   ( N-1  ---  N-2,FLAG-1  )
   BOTSYMBOLS OVER 1+ U< OVER TOPSYMBOLS U< AND ;

: ?FENCE   ( N-1  ---  N-2,FLAG-1  )
   DUP FENCE @ U< ;



-->
```

```
SCR # 22
( SYMBOLDICTIONARY            KS  10-5-80 )
: FORGET-SYMBOLS
   VOC-LINK @
   BEGIN DUP @ >R 2 - DUP >R @
         BEGIN BEGIN ?SYMBOL
               WHILE PFA LFA @
               REPEAT DUP R> !
               BEGIN PFA LFA DUP @
                     ?SYMBOL SWAP ?FENCE ROT OR 0=
               WHILE SWAP DROP
               REPEAT SWAP >R ?FENCE
         UNTIL
         DROP R> DROP R> -DUP 0=
   UNTIL
   MAIN-DIC BOTSYMBOLS DPSAVE ! ;
-->
```

```
SCR # 23
( SYMBOLDICTIONARY            KS  10-5-80 )

0 VARIABLE HEADFLG

: COMPILE-HEADS   ?EXEC 0 HEADFLG ! ;

: DROP-HEADS   ?EXEC 1 HEADFLG ! ;

: (CFA)
    0 , HERE 0 , IMMEDIATE
    DOES>   @ STATE @
    IF , ELSE EXECUTE THEN ;

-->


```

```
SCR # 24
( NEW CREATE             KS  10-5-80  )
: CREATE   HEADFLG @
   IF ?MAIN-DIC
      IF 2 ELSE SYMBOL-DIC 1 THEN
      HEADFLG !
   THEN
   TOD @ HERE 0A0 + U< 2 ?ERROR
   -FIND IF DROP NFA ID. 4 MESSAGE CR THEN
   HERE DUP C@ WIDTH C@ MIN 1+ ALLOT
   DP C@ 0FD = ALLOT
   DUP 0A0 TOGGLE HERE 1 - 080 TOGGLE
   LATEST , CURRENT @ ! HEADFLG @ 1 =
   IF 0 HEADFLG ! (CFA) 1 HEADFLG !
      MAIN-DIC DP C@ 0FF = ALLOT HERE SWAP !
      TOD @ HERE 0A0 + U< 2 ?ERROR
   THEN HERE 2+ , ;  -->
```

```
SCR # 25
( MOVE-DEF? I.E. THROW AWAY THE <BUILDS-PART    KS  10-5-80  )
: (;COD)
   LATEST PFA HEADFLG @ 1 = IF @ ELSE CFA THEN ! ;

: (;CODE)     R> (;COD) ;
: ((;CODE))  R> @ (;COD) ;

: MOVE-DEF?
   HEADFLG @ 1 =
   IF SYMBOL-DIC LATEST PFA DUP CFA DP !
      @ DPSAVE @ >R DUP DPSAVE ! R> OVER - DUP >R
      HERE SWAP CMOVE R> ALLOT COMPILE ((;CODE))
      HERE 2 ALLOT MAIN-DIC HERE SWAP !
   ELSE COMPILE (;CODE)
   THEN ;
-->
```

```
SCR # 26
( REDEFINITION OF <BUILDS   DOES>     GRL,KS  10-5-80 )

: <BUILDS
   CREATE SMUDGE ;

: DOES>
   MOVE-DEF?  020 C, [ HERE 8 + ] LITERAL , ; IMMEDIATE
   ASSEMBLER
   PLA, TAY, PLA, N STA, INY, 0=
   IF, N INC, THEN,
   IP 1+ LDA, PHA, IP LDA, PHA,
   IP STY, N LDA, IP 1+ STA,
   2 # LDA, CLC, W ADC, PHA,
   0 # LDA, W 1+ ADC, PUSH JMP,

-->
```

```
SCR # 27
( REDEFINITION  OF  ;CODE    KS  10-5-80 )

;CODE
   ?CSP MOVE-DEF? [COMPILE] [ SMUDGE
   !CSP [COMPILE] ASSEMBLER ; IMMEDIATE

;S



```

```
SCR # 28
( GOTO             KS  10-7-80 )
FORTH DEFINITIONS HEX
DROP-HEADS
: (GOTO)
   DOES> DUP @ BEGIN -DUP
               WHILE DUP @ SWAP
                     HERE OVER - SWAP !
               REPEAT HERE OVER !
               CFA [ ' 0 CFA @ ] LITERAL SWAP ! ;

: MOVE-HEAD   (  ---  HERE IN MAIN-DIC-1  )
   HERE SWITCH-DIC DUP HERE
   OVER C@ WIDTH C@ MIN 1+ DUP >R CMOVE
   HERE DUP 080 TOGGLE R> ALLOT DP C@ 0FD = ALLOT
   HERE 1 - 080 TOGGLE LATEST PFA LFA DUP @ , ! ;
-->
```

```
SCR # 29
( GOTO             KS  10-7-80 )
COMPILE-HEADS
: GOTO
   COMPILE BRANCH -FIND
   IF DROP DUP CFA @ [ ' 0 CFA @ ] LITERAL =
      IF @ HERE - ,
      ELSE [ ' (GOTO) 2+ @ ] LITERAL OVER CFA @ =
           IF BEGIN DUP @ WHILE @ REPEAT
              HERE SWAP ! 0 ,
           ELSE 4 ERROR
      THEN THEN
   ELSE MOVE-HEAD [ ' (GOTO) 2+ @ ] LITERAL , , SWITCH-DIC 0 ,
   THEN ; IMMEDIATE
-->


```

```
SCR # 2A
( GOTO             KS  10-7-80 )

: LABEL  -FIND
   IF DROP CFA DUP @ [ ' (GOTO) 2+ @ ] LITERAL =
      IF EXECUTE ELSE 4 ERROR THEN
   ELSE MOVE-HEAD [ ' 0 CFA @ ] LITERAL , , SWITCH-DIC
   THEN ; IMMEDIATE

FORGET-SYMBOLS
;S

( 28 - 2A TAKES 318 BYTES )




```

FORTH IN PRINT

THE TAYLOR REPORT/Alan Taylor

Alternative Software Making Great Strides

Imagine having your own private Cobol compiler -- with special security features and your user application statements – that you could develop and keep running on your future as well as current hardware. *That* would be a change indeed for any user, and as yet it is still just a dream. But there appear to be no technical reasons and few practical reasons to expect that such a compiler won't be generally available in a year or two.

The Forth Interest Group's (FIG) recent conference showed continued breakthroughs in really opening up software capabilities to users on at least six distinct fronts – hardware, languages, environments, cross-compiling, research targets and user training. This, only a year *after* the publication of the first FIG models of the Forth language, showed how some basic knowledge can bear fruit.

The power behind these and other developments has been a growing international group of people and firms. Headed by Chuck Moore's own Forth, Inc., independent user groups in America, Europe and Japan who appreciated the power of Forth have resulted in small commercial ventures with Forth compilers on micros. (The leader here, with more than 100 user groups of its own, is Miller Microsystems, located just a mile from *Computerworld's* headquarters!)

From this base of people, FIG is able to produce a technical journal, *Forth Dimensions*, which is improving all the time.

Since Forth is extendable -- that is any user can add new statements (either because the language is becoming more appreciated or else because the particular application or installation wants a different vocabulary) the journal's emphasis is on comparing different methods of implementing language elements. This focus allows the community to see how to keep the language efficient.

The journal also promotes the continued development of the Forth standard, annual conferences, and general communication among the many groups.

All this, however, is only as important as *what is actually made with* the FIG Forths. And that was why the 1980 conference was particularly important.

Outside Language

Forth, before now, had an outside language which, while somewhat Pascal-like, was distinctly forbidding and, because of the rareness of Forth programmers, something that users hated to use.

However, other more popular and conventional languages including Pascal, Lisp, Basic and (potentially) Cobol can be written in Forth, thus releasing the employment problem, while adding for their users the extending [illegible]

(Continued on Page 31)

FIGFORTH, TOO

If you have had a long wait for delivery of an order from the **FORTH** Interest Group (again in October 13's "Data Files"), it may be the post office's fault. I, too, ordered copies of the figFORTH manuals and source code for FORTH. It took 22 days for our super-efficient postal service to deliver my copies. Also received from the FORTH Interest Group were copies of *FORTH Dimensions*, its bimonthly publication. The September/October 1980 issue, larger than normal with over 90 pages, was professionally prepared and made good reading. The reason for this extra-size issue (regular issues seem to run about 35 pages) was publication of the source code for entries in a "CASE" statement contest. *FORTH Dimensions* is sent as part of membership in the FORTH Interest Group. The current membership cost is $12 per year in the U.S. and Canada, and $15 per year overseas.

RENEW NOW!

Forth for Alpha Micro's AMOS

PALO ALTO, CA — Professional Management Services' (PMS) Version 3.2 of µA/Forth, a fig-Forth (Forth Interest Group) product, is aligned with the 1978 standard of the Forth International Standards Team and allows complete access to Alpha Microsystems' multitasking operating sytems, AMOS.

Forth was developed for control applications, data bases, and general business. µA/Forth implements full-length names up to 31 characters, extensively checks code at compile-time with error reporting, contains string-handling routines and a string-search editor, and permits sealed vocabularies to control user access. Included is a Forth assembler, permitting structured, interactive development of device handlers, speed-critical routines, and linkage to operating systems or to packages written in other languages.

As an extensible, threaded language Forth words (commands) may be created from previously defined words, and even the original words supplied with the system (about 100) can be redefined if desired, adapting the language for special circumstances.

The distribution disk is in single density, AMS format, and includes all source code. The diskette includes an editor, a Forth assembler, and string package in Forth source code. This complete system is available for $130.

For additional information, contact Professional Management Services, 724 Arastradero Rd., Suite 109, 94306, (408) 252-2218. Circle 202.

RENEW TODAY!

MEETINGS

How to form a FIG Chapter:

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

2. Send to FIG in San Carlos, CA 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).

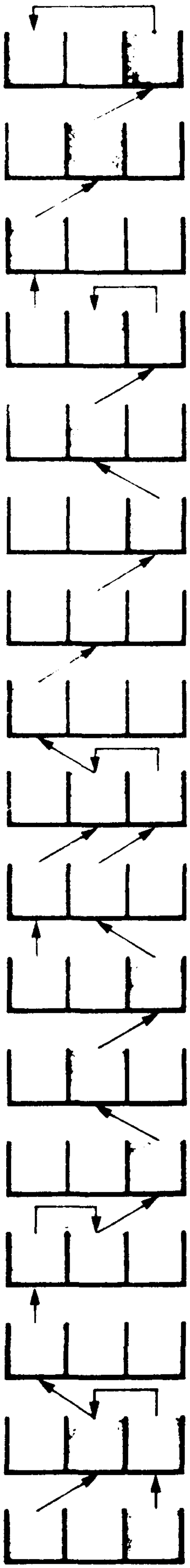
3. FIG will print, address and mail to members with the ZIP's you want from San Carlos, CA.

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 Saturday	FIG Monthly Meeting, 1:00 p.m., at Southland Shopping Ctr., Hayward, CA. FORML Workshop at 10:00 a.m.
Southern California	
4th Saturday	FIG Meeting, 11:00 a.m. Allstate Savings, 8800 So. Sepulveda, L.A. Call Phillip Wass, (213) 649-1428.
FIGGRAPH	
2/14/81 3/14/81	FORTH for computer graphics. 1:00 p.m. at Stanford Medical School, #M-112 at Palo Alto, CA. Need Info? L. Pearlmutter 415/856-1234
Massachusetts	
3rd Wednesday	MMSFORTH Users Group, 7:00 p.m., Cochituate, MA. Call Dick Miller at (617) 653-6136 for site.
San Diego	
Thursdays	FIG Meeting, 12:00 noon. Call Guy Kelly at (714) 268-3100 x 4784 for site.
Seattle	
Various times	Contact Chuck Pliske or Dwight Vandenburg at (206) 542-8370.
Potomac	
Various times	Contact Paul van der Eijk at (703) 354-7443 or Joel Shprentz at (703) 437-9218.
Texas	
Various times	Contact Jeff Lewis at (713) 729-3320 or John Earls at (214) 661-2928 or Dwayne Gustaus at (817) 387-6976. John Hastings (512) 835-1918
Arizona	
Various times	Contact Dick Wilson at (602) 277-6611 x 3257.
Oregon	
Various times	Contact Ed Krammerer at (503) 644-2688.
New York	
Various times	Contact Tom Jung at (212) 746-4062.
Detroit	
Various times	Contact Dean Vieau at (313) 493-5105.
Japan	
Various times	Contact Mr. Okada, President, ASR Corp. Int'l, 3-15-8, Nishi-Shimbashi Manato-ku, Tokyo, Japan.
Quebec, Canada	
Various times	Contact Gilles Paillard (418) 871-1960.

Publishers Note:

Please send notes (and reports) about your meetings.



FORTH DIMENSIONS

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

Volume II
Number 6
Price $2.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No 6 March April 1981

Publisher Roy C Martens

Guest Editor C J Street

Editorial Review Board

Bill Ragsdale
Dave Boulton
Kim Harris
John James
Dave Kilbridge
Henry Laxen
George Maverick

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL 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 $12.00 per year ($24.00 overseas 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 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 for 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 2,800 worldwide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

EDITOR'S COLUMN

The theme of this month's FORTH DIMENSION is practical applications.

During the last two years or so I have heard from many FIG members who seem to have a common problem— "Now that I have FORTH, where do I go from here?" In addition, many of us seem to be re-inventing code that others have already running, just because we are unaware of its existence.

In short, FIG members are suffering from a common problem—failure to communicate. Fortunately, this is an easily cured problem. FORTH DIMENSIONS is our communications vehicle; all we have to do is use it.

The mechanics are simple. FORTH DIMENSIONS is seeking short universal tool type code segments for publication. If you have some code that you have found especially useful and can explain its function and use, please contact the editor at FORTH DIMENSIONS.

YOU DON'T HAVE TO BE A WRITER! You will be sent a publication kit that leads you through the writing process. You will also be given all the help necessary by the FORTH DIMENSIONS editorial staff.

FIG members already have a reputation as creative problem solvers, now if we will just share and exchange our ideas, the permutations of that process boggle the mind. I am looking forward to enthusiastic response to this new approach that will benefit all.

C. J. Street

PUBLISHER'S COLUMN

It's the end of the FIG year and renewals are piling in. (Have you renewed?). Some of our newer members might be confused about renewing. If you recently joined FIG and received back issues of Volume II of FORTH DIMENSIONS then it is time to renew for Volume III and your March 1981 to March 1982 membership.

A number of other items of interest:

- FIG now has over 2800 members, worldwide
- FIG will have booths at the Computer Faire, April 3-5 in San Francisco and at the Jersey Computer Show in Trenton on April 25.
- There are a number of new listings — see order form at back
- Several reports from new chapters — lets see more
- Proceeding of 1980 FORML Conference is now available — see order form
- Looks like this is going to be our biggest year

Roy Martens

FORGIVING FORGET

Dave Kilbridge

Acknowledgment

I want to describe a FORTH system word which has come to be known as "smart FORGET" or even "Dave Kilbridge's smart FORGET." But the ideas involved appear in the State University of Utrecht, The Netherlands' FORTH system at least as early as 23 May 1978. The code presented here is a straightforward adaptation to the FIG model.

The Problem

The principal function of FORGET is to reclaim memory by locating in the dictionary the next word in the input stream and resetting the dictionary pointer (DP) to the beginning of the definition of that word. To avoid destroying vital parts of the system, no FORGETting is allowed below the address stored in FENCE. In the "dumb FORGET" of the original FIG model (see Screen 72), this address check is made on line 8.

But merely truncating the dictionary, even at a safe place, is not enough. The dictionary has a linked-list structure which allows it to be searched. If a link is left pointing into the "never-never-land" beyond the new value of DP, then the system may crash the next time a dictionary search uses that link.

These links are of two types: (1) VOCABULARY words have a link to the latest word in the vocabulary they name. "Dumb FORGET" adjusts this link (line 9) to point to the latest word which you don't FORGET, but only for the CURRENT and CONTEXT vocabularies. (Line 7 verifies that these are the same; this test was thought to give some extra protection against crashing. Any vocabulary not in CURRENT or CONTEXT may be trashed. (2) CURRENT and CONTEXT themselves point to vocabularies. If you FORGET the name of the CURRENT vocabulary, or any word before it in the dictionary, you may crash.

The Solution

"Smart FORGET" overcomes these hazards so effectively that I have never crashed by doing a FORGET. This is made possible by linking all the VOCABULARY words in the system into another linked list, enabling them to be located. The head of the list is stored in VOC-LINK. See the figure for the various fields in a VOCABULARY word.

How It Works

Refer to the code on Screen 18. On line 7, the name-field-address of the next input word is located in the dictionary; this is the point at which the dictionary will be cut off. An error message issues if this address is below the contents of FENCE. This cutoff address is saved on the return stack, and the head of the vocabulary list is put on the parameter stack. Now everything is ready for the real work.

The BEGIN ... WHILE ... REPEAT loop on lines 9-10 runs through all VOCABULARY words above the cutoff address and unlinks each from the list. If any such vocabularies are found, both CONTEXT and CURRENT are pointed to FORTH. This removes any links described as type (2) above.

Now the outer BEGIN ... UNTIL loop on lines 11-13 runs through the remaining VOCABULARY words. For each such word, the loop on line 12 finds the highest word below the cutoff address in the corresponding vocabulary. The vocabulary head is then pointed to this word, thus fixing the links of type (1) above.

Finally, DP is reset to point to the cutoff address (line 14).

Improvements

Executing FORTH DEFINITIONS if any VOCABULARY word is found beyond the cutoff address is unnecessarily drastic. One could test CURRENT and CONTEXT and only change them if they point beyond the cutoff, but it's probably not worth the trouble.

Extensions

1. In systems which allow dynamic chaining of vocabularies, one must check whether a vocabulary chained to is beyond the cutoff address. If so, it is replaced by FORTH. (The Utrecht system does exactly that.)
2. In later versions of the author's PACE system, a base-page pointer is allocated for each new defining word. These are released by FORGET. This is done by comparing pointer values with the cutoff address and does not involve the vocabulary structure.

```
SCR # 72
  0 (  ',  FORGET,                                        WFR-79APR28 )
  1 HEX    3   WIDTH   !
  2 : '           ( FIND NEXT WORDS PFA; COMPILE IT, IF COMPILING *)
  3    -FIND  0=  0  ?ERROR  DROP  [COMPILE]  LITERAL  ;
  4                                       IMMEDIATE
  5
  6 : FORGET             ( FOLLOWING WORD FROM CURRENT VOCABULARY *)
  7    CURRENT  @  CONTEXT  @  -  18  ?ERROR
  8    [COMPILE]  '  DUP  FENCE  @  <  15  ?ERROR
  9    DUP  NFA  DP  !  LFA  @  CURRENT  @  !  ;
 10
 11
 12
 13  -->
 14
 15
```

```
SCR # 18
  0 (  Smart FORGET                                   DJK-WFR-79DEC02 )
  1 : '           ( FIND NEXT WORDS PFA; COMPILE IT, IF COMPILING *)
  2    -FIND  0=  0  ?ERROR  DROP  [COMPILE]  LITERAL  ;
  3                                       IMMEDIATE
  4 HEX
  5
  6 : FORGET                       ( Dave Kilbridge's Smart Forget )
  7    [COMPILE]  '  NFA   DUP  FENCE  @ u<  15  ?ERROR
  8    >R  VOC-LINK  @  ( start with latest vocabulary )
  9  BEGIN  R  OVER  u<  WHILE   [COMPILE]  FORTH  DEFINITIONS
 10     @  REPEAT  DUP  VOC-LINK  !  ( unlink from voc list )
 11  BEGIN  DUP  4  -       ( start with phantom nfa )
 12     BEGIN  PFA  LFA  @  DUP  R  u<  UNTIL
 13     OVER  2  -  !  @  -DUP  0=  UNTIL  ( end of list ? )
 14   R>  DP  !  ;     -->
 15 This replaces Screen 72 of the F.I.G. Model.
```

SOME NEW EDITOR EXTENSIONS

Kim Harris

This article shows how to add two new commands to the FORTH editor which permit the replacement or insertion of multiple lines of a screen. This is a mini-application which demonstrates string input and output, adding new commands to the Forth editor, manipulating vocabularies, and a "terminal input processor" which prompts for input then processes it. Several variations in implementation are shown to illustrate different styles and refinements. If you are only interested in the final result, you can type in Screen 45 (in this article) into any standard fig-FORTH system which already has the FIG line editor (from screens 87 to 91 in the Installation Manual).

The use of the new commands will be illustrated by an example. Input is underlined; output is not. The symbol (CR) means to push the Carriage Return key (or equivalent).

To begin any editing of screen 100 you say

```
 100 LIST  EDITOR     (CR)
0   ( TEST SCREEN )
1   old 1st line
2   old 2nd line
3   old 3rd line
      . . .
```

To replace one or more lines starting at line 2, say

```
 2 NEW (CR)
0   ( TEST SCREEN )
1   old 1st line
2   -
```

The cursor is at the start of line 2 and waiting for you to enter new text. If you enter some text and a (CR), it will prompt you for a new line 3 and so on. This continues until you replace line 15 or enter only a (CR) at the start of a line. Then that line and any remaining ones are listed unchanged.

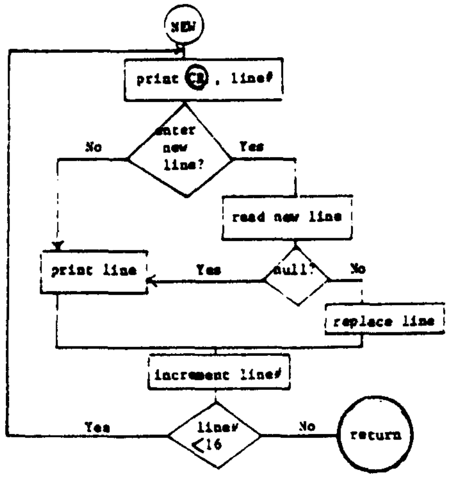
```
 2 NEW (CR)
0   ( TEST SCREEN )
1   old 1st line
2   new text for line 2 (CR)
3   something for line 3 (CR)
4  (CR) old 4th line
5   old 5th line
      . . .
```

A similar command UNDER lets you insert one or more lines starting at a specified line number.

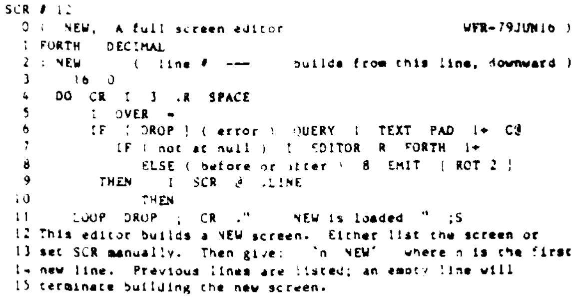
```
 2 UNDER (CR)
0   ( TEST SCREEN )
1   old 1st line
2   new text for line 2
3   inserted line (CR)
4   another inserted line (CR)
5  (CR) something for line 3
6   old 4th line
7   old 5th line
      . . .
```

Any lines pushed off line 15 are lost.

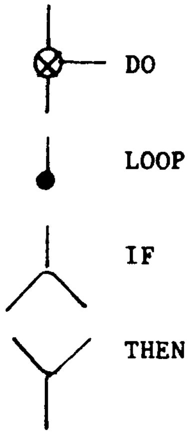
Let's design this application starting from the top. First consider the control flow for NEW and draw a flowchart. The one below is a traditional ANSI standard one.

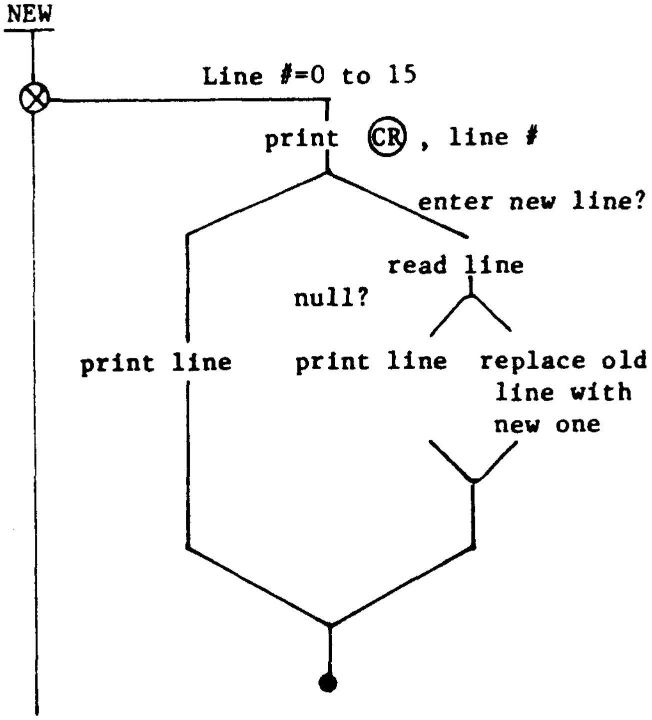


This flow chart is poor. It is unstructured (i.e., "print line" is improperly shared by two IF structures), the loop structure requires two boxes which can be performed by the single word DO, and no symbol exists for the word LOOP. To program this flowchart, you either have to cheat or change the flowchart. An example of cheating is in Screen 12. This implementation of NEW is by Bill Ragsdale and works fine. The tricks are the words inside square brackets on lines 6 and 8. These are manipulating the stack at compile-time, modifying the compiled branch structures. Such tricks reduce readability and modifiability, increase complexity, are neither "standard" nor transportable to non-FIG systems, and are not necessary.



```
SCR # 12
  0 (  NEW,  A full screen editor                         WFR-79JUN16 )
  1 FORTH   DECIMAL
  2 : NEW       (  line #  ---      builds from this line, downward )
  3      16  0
  4   DO  CR  I  3  .R  SPACE
  5       I  OVER  =
  6       IF  [ DROP ] ( error )  QUERY  1  TEXT  PAD  1+  C@
  7          IF ( not at null )  I  EDITOR  R  FORTH  1+
  8             ELSE ( before or after )  8  EMIT  [ ROT 2 ]
  9        THEN    I  SCR  @  .LINE
 10             THEN
 11     LOOP  DROP  ;  CR  ."     NEW is loaded  "  ;S
 12 This editor builds a NEW screen.  Either list the screen or
 13 set SCR manually.  Then give:   'n  NEW'   where n is the first
 14 new line.  Previous lines are listed; an empty line will
 15 terminate building the new screen.
```





Let's try modifying the flowchart to make it structured. Repeating "print line" under the 2 top decision boxes makes this proper. A different kind of flowchart prevents this kind of error and is ideally suited to FORTH. It is called D-charts and was described in FORTH DIMENSIONS, Vol. 1, No. 3. Not only is a D-chart inherently structured, but also there is a one-to-one correspondence between the chart symbols and FORTH words. In the D-chart of NEW, the correspondence between symbols and words is as follows:

We will certainly want to use as much of the existing editor as we can to reduce our work. The line Replace and Insert commands are good candidates:

R **line# -**
Replace line with text from PAD.

I **line# -**
Insert the text from PAD at line line#, old line line# and subsequent lines are moved down. Line 15 is lost.

We can use FORTH as a Program Design Language (PDL) by:

1) starting with the top word (e.g., NEW or UNDER),

2) making up names for lower words (i.e., forward references),

3) and using the postfix order and FORTH control structures but not worrying about correct stack manipulation.

Later the result can be finished by defining all the words used, supplying necessary stack manipulation operators, and typing them in and debugging each in bottom-up order.

From the previous D-chart we could write the following pseudo-definition for NEW:

```
  : NEW             16  0
DO
          CR   .LINE#
           ENTER? IF
     ENTER  NULL? IF
       .LINE   ELSE
    (EDITOR's) R  THEN
                  ELSE
          .LINE   THEN
                     LOOP
  ;
```

This incomplete definition does not take care of passing data on the stack or switching vocabularies. Look at the other command UNDER. The only change needed to the above code is to use the EDITOR's I instead of R. Because the two definitions are so similar, we will want to share some of the common parts.

To finish the definition of NEW, let's consider each undefined word.

.LINE#

needs to print the current line number right justified in 3 columns followed by a space. But should the line# be passed as a stack argument? The following definition sets it from the stack:

```
; .LINE# ( line# - ) 3 .R
SPACE ;
```

The FORTH word I could be used before the reference to .LINE# in NEW's definition to supply the DO-LOOP index (which is the current line number). But what about using I inside .LINE#'s definition instead? Unfortunately it's not the same. In fig-FORTH DO keeps its indices on the return stack, so I doesn't return the index in another definition even though it was called from a DO-LOOP body. Another word which does that is called I' (pronounced I prime). Then .LINE# could be written:

```
: .LINE#  ( - )  I' 3 .R
   SPACE   ;
```

A high level definition for I' is:

```
: I'
  FORTH R> R> R ROT ROT >R >R  ;
```

(A CODE definition would be preferred.)

Considering the inefficiency of I' and readability, let's pass the line number on the stack.

The next choice is should we use a separate definition for .LINE# (as above) or copy the contents of its definition into NEW. Execution speed would be indistinguishable. Using the name .LINE#

might be more readable, but not much. The dictionary sizes are different for the two choices. (Sizes are in bytes.)

	.LINE# separate		included in NEW & UNDER
literal 3		4	2 x 4 = 8
.R SPACE		4	2 x 4 = 8
.LINE# head	5 + name size=	10	
;		2	
references	2 x 2 =	4	
		24	16

So for only 2 references to .LINE#, it doesn't pay to define it separately. (3 references would make it close: 24 to 26 bytes.)

ENTER?

This should be true:

1) when the current line # equals the starting line #

2) while new text is being entered

3) but not after a (CR) only has been entered.

We never want to use a VARIABLE for temporary storage if we can help it. The starting line number comes in from the stack, so (1) is simple

```
start-line#  I  =
```

(The argument must be preserved each iteration, so a DUP must be added; a DROP will have to follow LOOP to compensate.) Case (2) can be achieved by incrementing the start-line# while in enter-mode. This can be done with a 1+ after the Editor's R. Finally (3) falls out by not incrementing it after either .LINE in NEW's definition.

Passing I on the stack would make ENTER? look like:

```
: ENTER? (start-line# current-line#-)
     OVER =  ;
```

But more words are needed in NEW's definition to complete the enter-mode control. As with .LINE# before, the contents of ENTER? could be copied in NEW's definition instead of being defined separately. The size tradeoffs would favor that, but in this case readability would be greatly enhanced by keeping the name. This also eliminates the need to comment each part of that IF structure (as in the version on Screen 12).

ENTER

must wait for terminate input, then copy the entire line to PAD for later use by the editor.

QUERY reads a line of input, and TEXT can copy it to PAD:

TEXT c -
Copy text from the Terminal Input Buffer to PAD until the delimiter c is found.

So we could define ENTER with:

```
: ENTER  ( - )
   QUERY  1 TEXT  ;
```

NULL?

should be true only if a (CR) was ENTERed. fig-FORTH puts a null character (i.e., binary zero byte) in the Terminal Input Buffer (TIB) when a (CR) is entered. To tell if it is at the start of the buffer, we can use:

```
: NULL? ( - f )
  TIB @  C@  0= ;
```

Although keeping this definition separate would take up more space than using its contents inside NEW and UNDER, readability is improved, so we'll keep it.

Finally, .LINE

needs a screen number and line number. The line number can be supplied by the DO-LOOP index. So before each .LINE in NEW or UNDER add:

```
I   SCR @   .LINE
```

Incorporating all the above refinements into the previous pseudo-definition of NEW produces the following code:

The only remaining changes needed concerns vocabularies. To add these definitions to the EDITOR vocabulary, use the phrase

```
EDITOR DEFINITIONS
```

before the first definition, and the phrase

```
FORTH DEFINITIONS
```

after the last. But within NEW's definition we need to specify which I and R are intended. FORTH uses pairs of names to resolve such ambiguities. It's like last names in people's proper names:

JOHN DOE

JOHN DEERE

But in good postfix style, the vocabulary name must precede the word it applies to, and remains in effect until changed. Vocabulary names in fig-FORTH are IMMEDIATE, so they can be used inside definitions the same way as outside. Within NEW's definition, we need to insert FORTH before DO to make sure all the I's are DO-LOOP words and not editor words.

```
: ENTER?   ( start-line# current-line# - f )   OVER =   ;

: ENTER   ( - )   QUERY   1 TEXT   ;

: NULL?   ( - f )   TIB @   C@   0=   ;

: NEW   ( start-line# - )                      16 0 DO

                               CR   I 3 .R   SPACE

                                       I ENTER? IF

                               ENTER   NULL? IF

                       I   SCR @   .LINE   ELSE

               I   ( EDITOR's )  R   1+    THEN

                                              ELSE

                          I   SCR @   .LINE   THEN

                                                  LOOP

       DROP   ;
```

Also we need to put EDITOR before the R (the editor's Replace command), and FORTH after R to make the remaining I's be DO-LOOP words.

Adding the vocabulary names makes the previous definitions testable. Trying them reveals that it all works except the line printed after the (CR) only was entered (i.e., leaving enter-mode) has one additional space before it. This skews that line from all the others. This is because fig-FORTH echos a space when the (CR) is entered. To fix this ugliness, back up the cursor 1 column before printing that line. For most terminals, a Back Space character will do the trick. (Not so on a memory-mapped terminal.) Defining the following will output a Back Space:

```
: .BS   ( - )   8 EMIT   ;
```

It should be inserted after the phrase NULL? IF in NEW's definition. Because this function is terminal-dependent, it definitely should be a separate definition.

The final working version follows:

```
SCR # 45
  0 ( EDITOR EXTENSIONS: NEW UNDER   KRH 9FEB81 )
  1 EDITOR DEFINITIONS
  2 : ENTER?   ( start-line# current-line# - f )   OVER =   ;
  3 : ENTER   ( - )   QUERY   1 TEXT   ;
  4 : NULL?   ( - f )   TIB @   C@  0=   ;
  5 : .BS   ( - )   8 EMIT   ;
  6
  7 : NEW   ( start-line# - )   FORTH   16 0 DO   CR   I 3 .R SPACE
  8    I ENTER? IF   ENTER   NUL? IF   .BS   I  SCR @  .LINE   ELSE
  9       I   EDITOR  R  FORTH  1+   THEN   ELSE   I  SCR @  .LINE
 10    THEN   LOOP   DROP   ;
 11 : UNDER ( start-line# - ) FORTH 1+  16 0 DO   CR   I 3 .R SPACE
 12    I ENTER? IF   ENTER   NULL? IF  .BS   I  SCR @  .LINE   ELSE
 13       I   EDITOR  I  FORTH  1+   THEN   ELSE   I  SCR @  .LINE
 14    THEN   LOOP   DROP   ;
 15 FORTH DEFINITIONS
```

NEW PRODUCT

HOME GROWN APPLE II SYSTEM:

As an avid FORTH user, I would like to share my work with other Apple II users. Assembling the fig-FORTH model source code on CP/M and other systems with assembly language development tools is relatively straight forward, but for the primarily turn-key Apple a lot of additional, undocumented information is required. To equalize this situation I will supply my home grown Apple II system on disk to anyone for $30.00. No documentation, support, or instruction is provided save for technical notes on the disk supplementing the FIG installation manual. An assembler, screen editor, source code and associated compiler are included. The idea is to be able to upgrade and patch the system in various ways from listings (standards, anyone?). Not for beginners, not a commercial product, at your own risk. Contact George Lyons, 280 Henderson St.; Jersey City, NJ 07302

TO VIEW OR NOT TO VIEW
(TO VIEW OR TO VIEW NOT?)

George William Shaw II

Sometime back, about one year ago, a fig-FORTH package was distributed to the members at the monthly FIG meeting. One of the programs in the package was a command called VIEW. This command would allow you to find the source text for a compiled definition and list it on the screen by simply typing VIEW, followed by the name of the command you wish to see the source text of.

I have been asked by Carl Street, the guest editor for this issue of Forth Dimensions, to write a commentary on this command which is to describe how the code originally submitted in the goodies package works and what other additions or changes I would make to the code.

So why have VIEW? VIEW adds convenience to writing and editing programs. The command allows you to get directly back to the source screen of a compiled definition, rather than trying to remember just what screen it was on. Most of us can remember approximately what screen or screens we have been working on, but if we have been working with more than a few screens, we would usually have to list a couple of screens to find the source to review or edit a given definition. VIEW eliminates this problem by allowing us to reference the source on the disk by the name of the compiled definition.

VIEW also takes very little system overhead. The entire compiled source for VIEW with all extensions mentioned in this article takes less than 170 bytes on my system. The compiling overhead is just as small. Only one or two bytes per definition and a negligible addition to compile time. Very inexpensive for the convenience and power it gives.

In order for VIEW to work, some of the resident defining words must be redefined. In pre-compiled fig-FORTH systems, the defining words CONSTANT, VARIABLE, VOCABULARY, : (colon), and <BUILDS must be redefined to contain a word called >DOC<. >DOC< will store in memory the disk screen number which contains the source of the definition being compiled. (On systems which can recompile themselves, >DOC< need not be placed in each one of the defining words. It need only be placed in the word CREATE, which is used by each of the defining words to enter a definition into the dictionary.)

With >DOC< in either CREATE or each of the defining words, the disk screen number which contains the source will be stored in memory to allow later referencing by the command VIEW. The command VIEW, then, has the task of finding the requested definition in the dictionary, fetching the screen number from memory, and listing the screen. This entire procedure is quite simple in FORTH and can be accomplished in a single line of source code (excluding the comment):

```
: VIEW       ( list source screen of definition )
        [COMPILE]  '  NFA  1  -  C@  LIST  [COMPILE]  EDITOR ;
```

The word [COMPILE] causes the word ' (tick) to be compiled into memory, rather than being executed at compile time (' is immediate). When VIEW is later executed ' will search the dictionary for the name which follows VIEW. NFA takes the address left on the stack by ' (the parameter field address) and changes it to the Name Field Address. 1 - then gives the address of the byte immediately preceding the name field. C@ extracts the screen number (which was stored at compile time) where the source of the definition is located. LIST prints the source of the definition. The second [COMPILE] allows EDITOR to be compiled (EDITOR is immediate) to select the editor vocabulary when VIEW is executed. This last step allows convenient entry into the editor for editing if desired.

```
    : >DOC<   BLK @ B/SCR  /  C,  ;
```

>DOC< stores a one byte screen number in memory of the screen from which source text is currently being interpreted or compiled. BLK contains the block number as above. In fig-FORTH, the block number and the screen number may not be the same (there may be several blocks per screen), so a division is performed with B/SCR (blocks per screen) to obtain the screen number. If in your system B/SCR is one (1), you may eliminate the division by B/SCR and additionally speed the execution of >DOC<.

```
: CONSTANT     >DOC<  [COMPILE]  CONSTANT   ;
: VARIABLE     >DOC<  [COMPILE]  VARIABLE   ;
: VOCABULARY   >DOC<  [COMPILE]  VOCABULARY ;
: :            >DOC<  [COMPILE]  :          ;
: <BUILDS      >DOC<  [COMPILE]  <BUILDS    ;
```

>DOC< is then placed immediately preceding each defining word to store into memory the screen number currently being interpreted. Since for most of us our fig-FORTH is pre-compiled (we can't recompile the basic FORTH system), each defining word is simply redefined to be preceded by >DOC<. The [COMPILE] in each of the words is actually only necessary in the redefinition of : (colon) because it is immediate and would attempt to execute at compile time rather than being compiled as desired. The other words are not immediate and would not have this problem.

Now, when any one of the defining words executes, >DOC< is executed, storing the screen number being compiled immediately preceding the name field of the definition. The area immediately preceding the name field was selected because this area can be addressed directly with existing FORTH words. The parameter field area of FORTH words is of variable length, so the area immediately following the end of the definition would not be as easily addressed.

When it is desired to VIEW a view-compiled word, the source screen number can easily be accessed and the definition listed. If a word which has not been compiled with the screen number preceding it is VIEWed, the screen determined by whatever byte immediately precedes the definition will be listed.

The current definition of VIEW works great except for a few minor idiosyncrasies. First, only a single byte is stored in memory for the source screen number. If you have screens above 255 and compile from them, the source cannot be viewed directly. A larger number is then needed. By simply changing the C, and C@ to , and @ in >DOC< and VIEW respectively, any screen currently accessible by the FORTH system could be VIEWed. Note that the address calculation must also be changed from 1 - to 2 - to account for the additional byte, as shown below:

```
: VIEW      ( list source screen of definition )
       [COMPILE]  '  NFA  2  -  @  LIST  [COMPILE]  EDITOR  ;
: >DOC<          BLK  @  B/SCR  /  ,     ;
```

Also, to the list of words being redefined I would add USER, CODE and CREATE. Redefining USER will allow the location of the definition of the user variable. Redefining CODE will allow the VIEWing of words defined in assembler. Redefining CREATE will cause all defining words later compiled to build VIEWable words.

```
: USER      >DOC<  USER    ;
: CODE      >DOC<  CODE    ;
: CREATE    >DOC<  CREATE  ;
```

It should be noted also that if you have changed the structure of your dictionary by placing links first (as I have) that the address calculation in VIEW will have to be changed as below:

```
: VIEW       ( list source screen of definition )
      [COMPILE]    ' NFA  4  -  @  LIST  [COMPILE]  EDITOR  ;
```

The additional 2 - (to 4 -) is necessary to skip the link which precedes (rather than follows) the name field in these systems.

And lastly, the current definition of VIEW will even try to list the source screen for definitions which have been created at the keyboard. The block number stored for these definitions is zero (0), which is not where the source is at all. If you don't mind having block zero (0) listed when you request to VIEW a definition which you created at the keyboard, then there is no problem. But, if this does bother you, you can put in the test below:

```
: VIEW      ( list source screen of definition )
      [COMPILE]  '  NFA  2  -  @
      -DUP IF   LIST [COMPILE] EDITOR   THEN   ;
```

In addition to the above, a test may be put in >DOC< to prevent the storing of the screen number when compiling from the keyboard:

```
: >DOC<        BLK  @  -DUP  IF  R> R      ,  THEN  ;
```

FIGURE 6

Note that if the test for block zero (0) is placed in >DOC<, then VIEW will try to list those definitions which would have had a screen number of zero (0) with the same result as attempting to VIEW a definition which was not defined with the redefined defining words.

George W. Shaw II
SHAW LABS, LTD.
P.O. Box 3471
Hayward, CA 94540

NEW PRODUCT

FORTH-79 FOR APPLE:

MicroMotion has announced the release of FORTH-79 for the Apple computer. MicroMotion FORTH-79 is a structured language that is claimed to conform to the new FORTH-79 International Standard. MicroMotion FORTH-79 comes with a screen editor and macro-assembler. Vocabularies are included for strings, double precision integers, LORES graphics and modem communication. Its operating system allows multiple disk drives and is 13 or 16 sector disk compatable. MicroMotion FORTH-79 runs on a 48K Apple II or Apple II Plus. Retail price is $89.95 including a professionally written tutorial and user's guide designed to make learning FORTH-79 easy for the beginner. MicroMotion; 12077 Wilshire Blvd., Suite 506; Los Angeles, CA 90025; (213) 821-4340

(Editor's note -- The manual is excellent. It notes the differences between fig-FORTH and FORTH-79 where pertinent)

RENEW TODAY!

SEARCH

John S. James

When you are debugging or modifying a program, it is often important to search the whole program text, or a range of it, for a given string (e.g., an operation name). The 'SEARCH' operation given below does this.

To use 'SEARCH', you need to have the FIG editor running already. This is because 'SEARCH' uses some of the editor operations in its own definition. The 'SEARCH' source code fits easily into a single screen; it is so short because it uses the already-defined editing functions. Incidentally, the FIG editor is documented and listed in the back of FIG's Installation Manual.

Use the editor to store the source code of 'SEARCH' onto a screen. Then when you need to search, load the screen. (Of course if you are using a proprietary version of FORTH, it may have an editor and search function built in and automatically available when needed. This article-ette is mainly for FORTH users whose systems are the ten-dollar type-it-in-yourself variety.)

Here is an example of using 'SEARCH'. We are searching for the string 'COUNT' in screens 39-41; the source code of 'SEARCH' is on screen 40. The screen and line numbers are shown for each hit. Incidentally, the search string may contain blanks. Just type the first screen number, the last screen number, SEARCH followed by one blank and the target text string. Conclude the line with return. The routine will scan over the range of screens doing a text match for the target string. All matches will be listed with the line number and screen number.

Happy SEARCHing!

Example of Use:

```
39 41 SEARCH COUNT
 00 VARIABLE COUNT_ER                                        2 40
    1 COUNT_ER +!    COUNTER @                               4 40
    1 COUNTER +!   COUNT_ER @                                4 40
    56 > IF   0 COUNT_ER !                                   5 40
    12 EMIT 01 TEXT   0 COUNT_ER !                           8 40  OK
```

CORRECTION:

CROMEMCO DISKETTES described on page 145 of Vol. II/5 are supplied by:

Inner Access Corp.
PO Box 888
Belmont, CA 94002
(415) 591-8295

GREATEST COMMON DIVISOR

Robert L. Smith

The problem of finding the greatest common divisor (GCD) of two integers was solved by Euclid more than 2200 years ago at the great library in Alexandria. The technique is known to this day as Euclid's Algorithm. The method is essentially an iteration of division of a prior divisor by a prior remainder to yield a new remainder. The quotients generated by this process are useful in other applications, such as rational fraction approximations, but are not required for finding the greatest common divisor.

For readers unfamiliar with the process, an example should clarify the method. Suppose we wish to find the GCD of 24960 and 25987. Divide one number into the other, and find the remainder or modulus:

```
25987 24960 MOD -> 1027
```

Divide the previous divisor 24960 by the remainder 1027 to yield:

```
24960 1027 MOD -> 312
```

Continue the process as follows:

```
1027 312 MOD -> 91

312 91 MOD -> 39

91 39 MOD -> 13

39 13 MOD -> 0
```

The last non-zero remainder is our desired answer, 13. This process must converge since the remainder is always less than the divisor. The process will terminate for finite numbers and integer division.

On Screen 20, we see a version of the greatest common divisor routine called G-C-D written in fig-FORTH. Line 1 begins a colon definition. In lines 2 and 3 the two arguments at the top of the stack are conditionally swapped to force the larger of the two arguments to be the first dividend. This step is used to avoid an unnecessary division in the succeeding part. However, lines 2 and 3 can be omitted entirely with no effect on the answer. The body of the calculation is in lines 4-7. At the start of the BEGIN-WHILE-REPEAT loop, the top element of the stack is the prior divisor and the second element is the prior remainder. In line 4 the new divisor (prior remainder) is saved, and the order of the top two elements reversed to prepare for the division. In line 5 the division with remainder is performed, and the remainder copied to the top of the stack for testing in line 6. For cases of non-zero remainders, the quotient is discarded but the remainder is kept in preparation for the next stage in the loop. The process terminates with a zero remainder. At line 8 the final quotient and remainder are dropped to yield the preceding remainder, which is the desired answer. Finally, the answer is printed out. The semicolon at the end of line 8 terminates the definition.

When Screen 20 is loaded, lines 9-11 are executed to print an invitation to the user to try the routine.

There are three areas in which this routine can be improved. The first is to remove lines 2 and 3 entirely, since the code does not usefully contribute to the final result. Furthermore, there is probably not even a speed advantage for machines with a hardware divide. Secondly, since the quotient is not used, the /MOD function can be replaced by the MOD function with a little reworking of the code. Finally, the printout function can be separated from the calculation function. It is usually advantageous in FORTH to write each definition so that it does as little as possible! The advantage of the separation in this case is that the calculation function can be applied repeatedly for finding the greatest common divisor of more than two arguments.

Our modified screen is shown below:

```
: GCD
     BEGIN
          SWAP OVER MOD ?DUP 0=
     UNTIL
     ;

: G-C-D
     GCD CR ." The G-C-D is " .
     ;

CR ." Input two numbers, then"
CR ." execute 'G-C-D'. The"
CR ." greatest common divisor"
CR ." of these numbers will be"
CR ." displayed."
CR
```

The sequence in GCD is quite easy to follow now. The two arguments on the stack are swapped, and the 2nd element is copied over the first, in preparation for the division implied by the MOD function. The word ?DUP is the 79-Standard version of the fig-FORTH word -DUP. The function of ?DUP is to duplicate the top element of the stack (the remainder from the division in this case), but only if the remainder is non-zero. The function 0= reverses the logical value of the top stack element, so that the test in UNTIL will cause a branch back to the BEGIN part when the MOD function results in a non-zero value. When the remainder is zero, the zero value is not duplicated. Instead, the 0= function converts it to a 1, which in turn is dropped by the action of UNTIL. Furthermore, control is then passed from the BEGIN-END loop, and the function terminates, leaving only the previous non-zero remainder.

Note that the number of FORTH words in the basic definition has been effectively cut in half, compared to the original version in Screen 20.

The author gratefully acknowledges discussions with LaFarr Stuart in the preparation of this article.

```
SCR # 20
 0 ( Greatest common divisor, a demo         WFR-79DEC09 )
 1 : G-C-D
 2    OVER OVER <
 3    IF  SWAP  THEN  ( use larger as quotient )
 4    BEGIN  SWAP  OVER     ( save divisor third )
 5           /MOD  OVER     ( test remainder zero )
 6      WHILE  ( not zero )  DROP  ( this dividend )
 7     REPEAT
 8         DROP  DROP  CR ." The G-C-D is " . ;
 9 CR ." Input two numbers, then execute 'G-C-D'.  The greatest"
10   ." common divisor of these numbers will be displayed."
11 CR
12
13
14 ;S
15
```

PROGRAMMING HINTS

PROGRAMMING APPLICATIONS, DEMONSTRATIONS & EXPLANATIONS

```
0        ( By guest writer Henry Laxen            Feb 81     )
1    HERE >R     ( Save current DP on return stack            )
2    : :: [ ' QUIT CFA @ ] LITERAL ,  ( Get runtime for :     )
3    !CSP ]      ( Security, start compiling                  )
4    BEGIN
5      INTERPRET   ( Compile what is typed                    )
6      STATE @ WHILE  ( Until state changes                   )
7      CR QUERY    ( Get another line from TTY                )
8    REPEAT
9    SMUDGE      ( Undo what ; did                            )
10   R> EXECUTE     ( Now do what user wanted                 )
11   R> DP ! ;      ( And restore dictionary                  )
12
13
14
15
```

:: is an excellent example of the flexibility of FORTH. Certain constructs in FORTH cannot be typed in from the terminal unless the user is in compilation mode. These constructs include: DO, LOOP, IF, ELSE, THEN, and all of the conditional compiling words. However, :: allows you to do this if you so desire. The idea is simple enough; you create an "orphan" word in the dictionary, execute it, and then forget it. (An orphan is a definition without a name header).

Let's step through the above definition line by line and see what is happening at each point:

1 HERE >R

HERE is the location of the next available dictionary entry. This location is saved on the return stack so it can be restored later.

2 [' QUIT CFA @] LITERAL ,

[changes from compilation mode to interpretation mode. QUIT has been previously defined as a high level : definition, and hence we use ' QUIT to get the address of its PFA. The CFA then converts this PFA to the CFA for QUIT. Since QUIT is a : definition, this CFA points to the runtime for : , which controls the nesting level in FORTH. The @ gets this address and places it on the parameter stack. Now the] places us back into compilation mode. The value we have thus computed, namely the runtime address of : , is then compiled as a literal in the definition for :: . When :: is executed, this literal is compiled inline by the , that follows. This has set up what follows as a : definition, so that it will execute properly when the time comes.

3 !CSP]

The !CSP is used for compile time error checking. The check is made when the user types ; to end his definition. The] puts the user into the compile state. What is typed from now on will be compiled instead of interpreted.

4 BEGIN

This denotes the beginning of some kind of looping structure.

5 INTERPRET

This is the main word in FORTH. It either executes or compiles the words it encounters depending on the current state. In our case it is compiling the words it encounters.

6 STATE @ WHILE

STATE is the variable which determines whether one is interpreting or compiling. When it is non-zero one is compiling, hence the loop is repeated as long as the user is still compiling. When you type ; STATE is set to zero, and this loop is exited.

7 CR QUERY

This simply gets another line from the terminal so that INTERPRET can compile it.

8 REPEAT

This is the end of the BEGIN loop.

9 SMUDGE

SMUDGE is used to undo the SMUDGE present inside of ; . It has no other purpose in this context.

10 R EXECUTE

This executes the word we have been building until now. If all goes well it will return.

11 R> DP !

And now we restore the dictionary to its previous state.

Note that there are still things you should not do with this implementation of :: , namely if what you are executing alters the dictionary, say by compiling additional words, the system will crash. An interesting exercise for the reader would be to redefine :: so that this is not the case.

This article contributed by Henry Laxen; 1259 Cornell; Berkeley, CA 94706

NEW PRODUCT

GO-FORTH FOR THE APPLE II:

The CAI FORTH Instruction System by Don Colburn is now available for the Apple. The GO-FORTH CAI System takes the novice FORTH programmer through the pitfalls of learning FORTH and lets him fly. Requires 48K Apple II plus Apple disc. Price is $45.00 per system (1-3 units), $30.00 per system (more than 4 units). International Computers; 110 McGregor Avenue; Mt. Arlington, NJ 07856; (201) 663-1580 (evenings).

NEW PRODUCTS

CROSS-COMPILER PROGRAM:

Nautilus Systems now offers a cross-compiler program for FORTH users. Machine readable versions are now available for the following hardware systems: LSI-11, CP/M, TRS-80, Apple, H-89, and Northstar. Each version includes: An executable version of figFORTH model 1.0; Cross-compilable source; Utilities; and Documentation. The cross-compiler is written entirely in high level figFORTH. Progam features include: Automatic forward referencing to any word or label; Headerless code production capability; ROMable code production capability; Load map; Comprehensive list of undefined symbols. Price is $150.00 including shipping. (California residents please add sales tax). Nautilus Systems; P.O. Box 1098; Santa Cruz, CA 95061; (408) 475-7461

TIMIN ENGINEERING FORTH:

Timin Engineering is now offering a version of figFORTH for 8080/8085/Z-80 or CDOS systems with at least 24K of memory. The Timin system features a FORTH style editor with 20 commands, a virtual memory subsystem for disk I/O, a Z-80/8080 assembler, and an interleaved disk format to minimize disk access time. Documentation includes a manual that may be purchased separately for $20 (credited towards purchase of disk). Price $95.00 on IBM compatible 8 inch single density disk (other disk formats $110) -- California residents please add 6% sales tax. -- and includes shipping by mail in U.S. Mitchell E. Timin Engineering Company; 9575 Genesee Avenue, Suite E-2; San Diego, CA 92121; (714) 455-9008

DEVELOPMENT OF A DUMP UTILITY

```
(DEVELOPMENT OF A DUMP UTILITY By John Bumgarner  March 81  )
OK
OK
 1  (DUMP MEMORY BYTES. ADDRESS COUNT. . .              )  OK
OK
 2  : DUMP  0 DO  DUP I + C@ 3 .R LOOP DROP ;  OK
OK
 3  HEX  OK
 4  1 2 3  HERE  10  DUMP  CR . . .  4 44 55 4D 50 20 20 20 20 20 20 20 20 20 20 20
 5  3 2 1  OK
OK
 6  ( Test for non-printing ASCII Character.  CH .. . T/F)  OK
OK
 7  : ?NON-PRINTING  DUP 20 <  SWAP 7E >  OR ;  OK
 8  : Q  ?NON-PRINTING . ;  OK
 9  -10 Q 0 Q 10 Q  1F Q  20 Q  40 Q 1 1 1 1 0 0  OK
10  7E Q 7F Q 100 Q  7FFF Q 8000 Q  0 1 1 1 1  OK
11  FORGET Q  OK
OK
OK
12  ( Type any memory bytes using  .  for non-printing characters)  OK
13  ( Address Count ...)  OK
OK
14  : &TYPE  0 DO  DUP I + C@
       DUP  ?NON-PRINTING IF DROP 2E ( .) THEN EMIT LOOP DROP ; OK
OK
15  1 2 3 HERE 10 &TYPE CR . . . .&TYPE
16  3 2 1  OK
OK
OK
17  (Print address , DUMP &TYPE 16 bytes.  Address ... )  OK
OK
18  : A-LINE  CR DUP 0 6 D.R SPACE 10 OVER OVER DUMP 2 SPACES &TYPE ;  OK
OK
19  HERE A-LINE
      2419  6 41 2D 4C 49 4E 45 20 20 20 20 20 20 20 20 20     .A-LINE  OK
OK
OK
20  2 3 4 HERE A-LINE CR . . .
      2419  6 41 2D 4C 49 4E 45 20 20 20 20 20 20 20 20 20 .A-LINE
4 3 2  OK
OK
OK
21  (Can't think of a better name.  Address count ... )  OK
OK
22  :  DUMP  0 DO DUP I + A-LINE 10 +LOOP DROP ; DUMP isn't unique OK
OK
23  HERE 40 DUMP
       243A  4 44 55 4D 50 20 20 20 20 20 20 20 20 20 20 20 .DUMP
       244A 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
       245A 20 20 32 84 44 55 4D DO 5F 5F 5F 5F 5F 5F 5F F5   2.DUM._          _
       246A 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F _               _  OK
OK
OK
```

Program development in FORTH can --and should--be done in a "top down" manner as this type of design produces error free programs in a minimum of time. However, the road to a solution is not always direct; and experienced FORTH programmers often play with a programming problem on the terminal to get ideas. These idea sessions usually result in the data necessary for "top down" program design. The development steps involved in producing a really useful tool from a very simple DUMP word will illustrate just such a design session.

I begin by setting down in the comment on line 1 the functions and parameters for the fundamental DUMP word to be defined -- I simply want it to dump values from memory. Given the starting address and number of bytes to dump as parameters on the stack, the word defined in line 2 is short and simple. The count value is on top of the stack which suggests using a DO ... LOOP to do the work. Inside the loop we put the code to generate an address, fetch and print a byte. The loop parameters are the count value on the stack and the zero put inside the definition just in front of the DO.

FORTH DO loops increment the loop index and check it against the limit at the end of the loop. If the newly incremented index is less than the limit, the loop body is executed again. If the index is greater than the limit, the loop is terminated and execution continues with the next word. This method of loop control results in the loop index (fetched for use by the word I) going from 0 to count 1- (that is Count-1 for non-FORTH persons) by 1's for this FORTH DUMP word. The loop index then is just what we need to add to the starting address to obtain successive byte addresses for dumping.

The inside of the loop first duplicates the address (the only thing remaining on the stack after DO removed its two arguments) to save a copy and then adds the loop index to it. Using this new address we fetch a byte with C@ and print it, right justified in a 3 column field using 3 .R . In the same manner we keep looping (incrementing the index by 1 each time) adding, fetching and printing. When the loop index equals the count, the loop exits and drops the extra copy of the address to clean up the parameter stack--A very important step!

FORTH is highly interactive and nothing is more natural than to test the new DUMP word immediately. In line 3 I switch to hexadecimal I/O to enable easy interpretation of the dump results in line 4. Before I actually execute DUMP in line 4 I put a few numbers on the stack so that I can check to see that the stack is not altered by DUMP . The word HERE provides a starting address for DUMP and then I select 10 (in decimal, 10 hex is 16) bytes to dump. The CR . . . after DUMP prints (on a new line) and removes the top three numbers of the stack. Lines 4 and 5 show the execution of the word and also show my check numbers are where they should be. This simple check shows that DUMP is very likely to be free of errors and we may proceed with confidence.

I happen to like dumps that dump both byte values and ASCII characters and so my next task was to make a word that could dump in ASCII what DUMP output as numbers. I have also learned from frustating experience that printers and terminals should not be given byte values that are non-printing characters lest they do some strange things in response. By thinking ahead a little bit I can see a definite need for a word to alter the memory byte values to an acceptable range for the output device. The result of this advanced thinking is shown on lines 6 and 7. Given a character value on the stack ?NON-PRINTING will return a True (plus one) value if it is a non-printing character and an False (zero) otherwise. The test is made for values less than hex 20 (a space) and greater than hex 7E (a tilde).

At this point I am still in hex for numeric I/O (see line 3) and so I use 20 and 7E directly in the definition without having to look up their equivalent decimal values. The OR at the end of ?NON-PRINTING combines the results of the two limit checks and produces a single true/false value for the output if either limit is exceeded.

The verification of a limit checking word like ?NON-PRINTING requires careful testing with data that exercises the word over its entire range and especially at critical places such as the limits. With ?NON-PRINTING this means testing 10 or more times with carefully chosen input values. The name of ?NON-PRINTING is nice and descriptive but is too long for a poor typist like me to type often so on line 8 I have defined the word Q to execute ?NON-PRINTING and print the result. On lines 9 and 10 I try my new Q word out with some numbers designed to make it fail if it will. Numbers such as small negatives, zero, small positives, on either side of the lower limit, mid range, either side of the upper limit, larger positives and at the magic place where the sign bit changes value. The results show that ?NON-PRINTING is not fooled by these tests and so on line 11 I discard my test word Q .

On lines 12 and 13 I put down my ideas for the word which will type the ASCII character equivalents for my memory byte values and begin the definition on line 14. This definition is just like the one for DUMP except for what happens to the byte value after it is fetched. In place of the 3 .R of DUMP we check for non-printing characters with ?NON-PRINTING and if they are non-printing we DROP the value and replace it with the hex code for a dot, 2E. Then we EMIT the byte to see the ASCII character for that byte's value.

Some programmers will want to mask the byte value with hex 7F to zero the high bit before testing for non-printing characters. One reason to do this is so fig-FORTH names will show up better in dumps since the "traverse" bit on the last kept name character would otherwise turn that character into a non-printing value and it would appear as a dot. My reason for not doing it is that getting rid of the high bit turns too much data into ASCII and clutters up that part of the dump. The testing of &TYPE is done exactly like the testing for DUMP , the .&TYPE after the CR . . . is the output of &TYPE . The dot before the ampersand is the count byte of the string getting turned into a dot by ?NON-PRINTING .

Incidentally, the string found at HERE when a given word is executing in the interpretive mode (outer or name interpreter in this case) is the name of the word. Look up the ASCII characters for the byte values dumped in line 4 and you will find they spell DUMP . This bit of magic is performed by the word WORD in figFORTH.

With the two words DUMP and &TYPE we are in a good position to write a word that will dump one line of information on the terminal. This word can then be executed in a loop, once for each line dumped. I call this word A-LINE and define it in line 18 after noting in line 17 that it should print the starting address of the dumped line. A normal line width on a CRT type terminal is 80 columns and so there is room for 16 bytes dumped (3 16 * columns), typed (16 more columns) and an address and spaces for separators (9 columns). The room used adds up to 73 columns but this could be reduced a few columns. If the address were always printed as an unsigned hex number it would only take up 4 spaces. Then by allowing only one space between the three fields on a line the line width would be 70 columns (4 1+ 48 + 1+ 16 +). If you have a smaller output line then you must give up the ASCII characters or the spaces between the byte values or dump less bytes per line.

The actual work done by A-LINE is to first do a CR to get a new line to use and then output the address as an unsigned number, right justified in a 6 column field. The address output is done by the phrase:

```
DUP  ( Save a copy of the address )
  0  ( Put a zero on top of )
     ( the stack to make a positive )
     ( 32 bit number )
  6  ( 6 column field )
D.R  ( Output a 32 bit number )
     ( in a field )
```

Next we output a SPACE to help separate the address from the byte values; put 16 (16 is hex 10) on the stack and copy both the saved address and our count of 16 by using OVER OVER (if you have it use 2DUP) . The stack is now set up with two sets of addresses and counts ready for our two words DUMP and &TYPE . I put 2 spaces in to separate the byte values from the ASCII characters.

On lines 19 and 20 of the printout I test A-LINE for stack problems and functionality and it seems that everything is working correctly.

At last I am now ready to write the actual dump word. The comment on line 21 starts me off with the desired parameters and a note showing my limited personal vocabulary. My inability to think of another name is not going to be a problem, however, since FORTH allows you to redefine names and use them over. All that happens is that a warning message (isn't unique) is displayed to alert you to the redefinition. The warning message appears here after the end of the definition on line 22. When the new definition with the re-used name successfully compiles, the earlier definition of the same name becomes unavailable for future use--either by compiling into new words or interpretively executed from the terminal. Since I do not want my old definition of DUMP on line 2 anymore, I consider this to be an advantage. I have an improved DUMP and I do not have to think of another name for it! What is more, everything will work properly as before because the use of the old DUMP in the definition of A-LINE does not get changed--what is compiled stays compiled as it was. All that happens as far as a user of DUMP is concerned is that if DUMP is now asked for, the new one will be found first automatically and the search will stop there -- the system never suspects that another, different version of DUMP is hiding down the dictionary a ways.

The actual definition of the new DUMP is very similar to the old DUMP except that we are substituting A-LINE for the C@ 3 .R and since we are dumping 16 bytes on a line we use +LOOP to terminate the DO and give it a 16 (10 hex) to add onto the loop index each time. Now the address computation done by the DUP I + phrase will start at the specified point and go up by 16 bytes each time through the loop.

The useful tool that I set out to develop at the start of this session is now complete and only needs to be tested. Line 23 does this final test, but because of my earlier successful test of A-LINE and also because the new DUMP is so similar to the old DUMP I did not bother to try putting some numbers on the stack to see if it adds or removes any values. I now have high confidence that DUMP will work correctly and it does.

This article contributed by John Bumgarner; FORWARD TECHNOLOGY; 1440 Koll Circle, Suite 105; San Jose, CA 95112

NEW PRODUCT

MICROPOLIS FORTH:

Acropolis now offers FORTH for Micropolis. Acropolis FORTH (A-FORTH) runs under Micropolis MDOS on 8080/8085 and Z-80 systems running at 2 or 4 MHz with 32K memory and at least one MOD I or MOD II disk. A-FORTH has 2 program/data file editors -- a line editor for standard serial terminals and a screen editor for memory-mapped terminals. A-FORTH has an 8080/8085 macro-assembler that allows use of any mixture of A-FORTH and assembly code desired in a single definition. A-FORTH has all the features of figFORTH plus: Double precision math & stack operations (32 bit); Double precision variables & constants (32 bit); Multi dimensional arrays up to the limits of available memory; Virtual arrays up to the limit of disk storage on all disks; Case statements; Printer support using MDOS ASSIGN statements; Forgetting across vocabulary boundaries; Enhanced disk procedures that reduce response time, compiling time, & number of disk accesses; Physical disk support for disk diagnostics and disk copy and direct access to MDOS file directory. Acropolis A-FORTH has an 89 page users manual. Acropolis provides A-FORTH updates & patches at no charge for 1 year after purchase. Price $150.00 including shipping (California residents add 6% sales tax). Acropolis division Shaw Labs, Ltd.; 17453 Via Valencia; P.O. Box 3471; Hayward, CA 94540; (415) 276-6050

NEW PRODUCT

ALPHA MICRO REENTRANT FORTH:

Sierra Computer Company is now offering version B of their AM-FORTH for Alpha Micro system AM-100 computers. Version B is said to be reentrant, allowing the basic FORTH dictionary to be loaded as part of the AMOS system and shared by any number of users in the multi-user Alpha Micro system. Other new features include: An assembler; Screen oriented editor; Support of special AMOS CRT handling features; Floating point math operations; Utilities for string handling and building data structures and access to system TIME and DATE functions; More versatile I/O to AMOS sequential and RANDOM files; and use of lower case characters. All features of version A are included in version B. AM-FORTH version B is available on AMS or STD disk that contains complete source code; executable object code; FORTH utilities for the editor, assembler and data structures and some sample FORTH programs. Complete documentaton describing AM-FORTH implementation, installation procedures, operating instructions and glossary. Price is $150.00 ($120.00 to licensed version A purchasers at $40.00). Contact George Young; Sierra Computer Company; 617 Mark NE; Albuquerque, NM 87123

LETTERS

Dear FIG,

I have been using the May '79 release of 6800 fig-FORTH since it was issued. A question that isn't apparent on the order form is, has a further release been made either on the assembly source listing or installation manual? This may be a saving for us by preventing a duplication of our software.

N.H. Champion
Prescott, AZ

Two small changes have been made to the FIG model. In screen # 23, U* has been corrected for a carry bug. Screens 93 and 94 have been converted from assembly to high level. The assembly listings have not changed in the last year. Here are the revision/publication dates for each FIG publication:

Publication	Release	Date
Installation Manual	1.0	11/80
Listing		
8080	1.1	9/79
6800	1.0	5/79
6809	1.0	6/80
6502		
9900	1.0	3/81
8086/88	1.0	3/81
PDP-11	1.3	1/80
PACE	1.0	5/79
ALPHA MICRO	1	9/80

Hope this clarifies your question. -- ed.

Dear FIG,

I would like to see programming examples as part of every meeting agenda. Perhaps a theme could be established for each meeting.

Also publishing programming examples would be helpful. I, for one, find examples the best method of learning and attempting to reach the point where I start building FORTH programs efficiently.

I also recognize that one person can't do it all, and that a successful users group depends upon contributions from everyone. I am not sure how I can help at this time, but I am willing to do my share.

J. Arthur Graham
Orinda, CA

Glad to hear it! See this month's edition for programming examples and this month's editor's column regarding helping out. -- ed.

Dear FIG,

I am interested in corresponding with others interested in FORTH on larger machines. I can be reached at the address below.

Stewart Rubenstein
HARVARD UNIVERSITY CHEMICAL LABS
12 Oxford St., Box 100
Cambridge, MA 02138

Dear FIG,

Would it be possible to include some tutorial articles on the inner workings of FORTH in FORTH DIMENSIONS?

Being new to this language, [I find] the functions of the interpreters and the compiler somewhat mysterious.

LETTERS

Given the extensibility of FORTH, a better understanding of the guts of the language is an advantage. I haven't found a publication that completely describes these functions.

R. Stockhausen
Milwaukee, WI

See Dr. C.H. Ting's SYSTEMS GUIDE TO fig-FORTH, available from FIG -- you can use the order blank at the back of this issue. -- ed.

Dear FIG,

Our membership is growing and I have delivered fig-FORTHs to several of the Black African countries.

Rhodes University has adopted 6809 fig-FORTH for its curriculum this year. UNISA will follow, God willing, next year in its microprocessor course, and several other universities are using various versions of fig-FORTH for research purposes.

I've written several articles, both local and abroad on FORTH and I'll send you copies of these. Please give us some support and coverage. I'll write you at least once a month.

Ed Murray
FORTHWITH COMPUTERS/FIGSA
P.O. Box 29452
Sunnyside, Pretoria, 0132
South Africa

Always happy to hear from our international contingent! Your meeting announcements are in our announcement section. -- ed.

DEA- FIG,

I AM A FOR-- PRO------- CUR------ EMP----- BY FOR-- INC. I HAV- NOT WOR--- ON ANY FIG TYP- SYS---- AND I AM EXC---- BY THE VAR----- LEN--- NAM- IDE-. PLE--- SEN- ME THE FIG FOR-- MOD-- SO THA- I MAY TRY IT OUT HER- AT FOR-- INC.

FRE- THO----

Your request is answered. Next edition you will be able to communicate with four+ letter words! -- ed.

Dear FIG,

I am a long time FIG member and am seriously devoted to FORTH as a programming language and system. Like many others, I have FORTH running now and after all the talk about how great it is, I find few (hardly any) complete examples of its use in solving real, practical problems.

The point of all this is a suggestion that FIG publish more articles and papers on practical applications -- programs which can be easily put into everyday use by any programmer. One can go to the magazine counter at any computer store and find many examples of practical programs in BASIC. FORTH should be even more appropriate for such applications.

I believe that the organization, and each of us as members, can contribute to this end. I propose FIG strongly solicit contributions of articles dealing with practical programming projects developed in FORTH.

George O. Young III
Albuquerque, NM

You took the words right out of our editorial mouths. We hear you and are looking forward to receiving contributions. -- ed.

Dear FIG,

I have developed a generalized data structure for vocabularies which removes many of the limitations now found in both FIG and other FORTH models.

My new structure has most of the advantages of the present FIG model, plus it allows multiple threads per vocabulary with different numbers of threads in each, if desired. With this multiple thread concept vocabularies are physically linked with a single pointer and are both sealed and linked simultaneously.

I have also developed a "vocabulary stack" to allow context specification in line with the FORTH '79 standard. I intend to make my findings available at the next FORML conference.

If anyone would like to contribute suggestions or developments along these lines (especially the vocabulary stack) for release in the public domain please write me at the address below:

EXIT (in line with the 79-standard)

George W. Shaw II
SHAW LABS, LTD.
P.O. Box 3471
Hayward, CA 94540

Dear FIG,

Help! A while back I got my copy of figFORTH-8080 version. I'm bogged down at the "MATCH" primitive of the "EDITOR" function. I'm working alone at it as home computers are rare up here and FORTH is a "Very Foreign Language". All I need to know is what in tarnation one uses to interpret screens 93 & 94 to 8080 (or Z80) code?

I have the rest of the FIG model working, although I've had moments with it ranging from tears to apoplexy. I've discovered some of the no-no's the hard way, also known as "How to reconfigure your disk -- unexpectedly." or "Where did the CP/M go?".

I haven't had so much fun since I built this "United Nations computer".

Regards,
Glenn Farnsworth
Weed, CA

Editor's note -- The editor was included with the model as an extra "goodie". A little foresight would have told us the 6502 assembly source would prove to be an irritant. The high level equivalent is given below. A full screen search with a code MATCH takes about 150 msec, while the high level form requires over a second. Try the high level version and then recode for your processor. This addition to the model was made in September 1980 thanks to Peter Midnight who provided an earlier definition.

Keep smiling! -- ed.

```
SCR # 148
 0 ( double number support                         WFR-80APR24 )
 1 ( operates on 32 bit double numbers or two 16 bit integers)
 2 FORTH DEFINITIONS
 3
 4 : 2DROP    DROP  DROP  ;                 ( drop double number )
 5
 6 : 2DUP   OVER  OVER  ;          ( duplicate a double number
 7
 8 : 2SWAP  ROT  >R  ROT  R>  ;
 9                   ( bring a second double to top of stack )
10 EDITOR DEFINITIONS  -->
11
12
13
14
15
```

```
SCR # 149
 0 (  String MATCH for editor                   PM-WFR-80APR25 )
 1 : -TEXT                    ( address-3, count-2, address-1 --- )
 2 SWAP  -DUP IF  ( leave boolean matched=non-zero, nope=zero)
 3               OVER  +  SWAP   ( neither address may be zero! )
 4          DO  DUP  C@  FORTH  I  C@  -
 5                IF  0=  LEAVE  ELSE  1+  THEN  LOOP
 6          ELSE  DROP  0=  THEN  ;
 7 : MATCH (cursor address-4, bytes left-3, string address-2,)
 8            (string count-1, --- boolean-2, cursor movement-1)
 9  >R  >R  2DUP  R>  R>  2SWAP  OVER  +  SWAP
10 (caddr-6, bleft-5, $addr-4, $len-3, caddr+bleft-2, caddr-1)
11  DO  2DUP  FORTH  I  -TEXT
12   IF  >R  2DROP  R>  -  I  SWAP  -  0  SWAP  0  0  LEAVE
13        ( caddr bleft $addr  $len  or else 0 offset 0 0      )
14    THEN  LOOP  2DROP  ( caddr-2, bleft-1, or 0-2, offset-1)
15   SWAP  0=  SWAP  ;
```

ANNOUNCEMENTS

PREVIEWS OF COMING ATTRACTIONS (IN FORTH DIMENSIONS):

Issue	Editorial Content
May/Jun	Applications, utilities & useable programs
Jul/Aug	Games & game type applications
Sep/Oct	University of Rochester & Utrecht conferences
Nov/Dec	Graphics & music

If you would like to be a contributing author to any of the above please write to: Editor; FORTH DIMENSIONS; P.O. Box 1105; San Carlos, CA 94070. You will be sent a writer's kit that will make your job easier. Please note deadlines for each issue are several months in advance of publication dates so allow plenty of time to produce your article.

FIG GOES TO COMPUTER FAIRE: FIG will have booth number 1137C at the West Coast Computer Faire being held April 3 to 5 at Brooks Hall in San Francisco.

FREE BUG FIXES: The 8080 Renovation Project wants bug reports so they can get to work on fixing them. If you have found an 8080 Bug send it to 8080 Renovation Project; c/o FORTH Interest Group; P.O. BOX 1105; San Carlos, CA 94070

DR. DOBBS NEEDS YOUR HELP: The editor of Dr. Dobb's Journal of Computer Calisthenics & Orthodontia is very interested in articles on FORTH. If he can get enough, he will devote an entire issue to FORTH. Interested authors should contact Marlin Ouverson, Editor; PEOPLE'S COMPUTER COMPANY; PO Box E; Menlo Park, CA 94025

CALL FOR PAPERS

FIG STANDARDS TEAM: The FORTH Standards Team announces the Spring Conference hosted by the University of Rochester on May 13th through May 15th, 1981. Larry Forsley is the session organizer. This conference will have three components: Formal papers, Sub-team working groups, and Poster sessions.

Formal papers must be received by May 1st. Later material and informal presentations will be assigned to the "Poster session"; at which the authors will conduct clustered workshops, with attendees moving among the presentations. The Sub-teams will prepare short reports after topic oriented working sessions.

Working sessions are scheduled from the morning of May 13th through lunch on May 15th. A reception will be held on the evening of May 12th for early arrivals. Accomodations are $12.00 single occupancy and $9.00 each, double occupancy. A combination of campus and off-campus meals are planned.

Papers are specifically requested on:

1. Implemtation aspects of FORTH-79
2. Refinements of vocabulary structure, extensible control structures, definition of input and output streams.
3. File sytem extension
4. Floating point extensions

The contact for submittal of papers and room reservations is Larry Forsley; Laboratory for Laser Energetics; University of Rochester, NY 14623. Send room requests without delay; a confirmation with exact cost will be returned with the conference schedule and travel suggestions.

MEETING/EVENT ANNOUNCEMENT FORMAT

In order to have uniformity and insure complete information in all meeting and special event announcements, FORTH DIMENSIONS requests that you use the following format:

1. WHO is holding the event (organization, club, etc.)
2. WHAT is being held (describe activity, speakers' names, etc.)
3. WHEN is it being held (days, times, etc.; please indicate if it is a repetitive event -- monthly meeting etc.)
4. WHERE is it being held (be as complete as possible -- room number, etc.)
5. WHY is it being held (purpose, objectives, etc.)
6. REMARKS & SPECIAL NOTES (is there a fee, are meals/ refreshments being provided, dress, tools, special requirements, pre-requisites, etc.)
7. PERSON TO CONTACT
8. PHONE NUMBER/ADDRESS (include area codes, times to call & give work & home numbers in case we need clarification)

ATTENTION 6502 USERS:

The following seem to me to be errors in the 6502 Assembly Source Listing (May 1980). I think I can correct these errors easily enough, but I worry if maybe they have generated more subtle errors that I have not found. I have no experience with FORTH at all, so I'm not sure what should be happening, and I have no one I know with any experience to call upon.

Page 0061 UPDATE Missing SEMIS at end? (There is one in the installation manual)

0064 Line 3075 Shouldn't this be a backward branch with F6 FF as displacement?

0067 Lines 3204 - 3205 Two STX XSAVE's. Is one superfluous or is it replacing something else that really should be there?

0069 Lines 3280 - 3284 Two SEMIS. Again, is something being destroyed by the extra one?

C.A. McCarthy
Department of Mathematics
Vincent Hall
UNIVERSITY OF MINNESOTA
Minneapolis, MN 55455

MEETINGS

FORTH INTEREST GROUP U.K.: Chairman: Dick de Grandis-Harrison; Secretary/Treasurer: Harry Dobson; Newsletter Editor: Gil Filbey; Committee Members: Bill Powell, Bill Stoddart. Meetings are held at 7 p.m. on the 1st Thursday in every even month at:

The Polytechnic of the Southbank
Room 408
Borough Road
LONDON

Mailing Address:

FORTH INTEREST GROUP U.K.
c/o 38, Worsley Road
Frimley, Camberley,
Surrey, GU16 5AU
ENGLAND

PORTLAND FORTH USERS GROUP: Held its first meeting in January. Demos were given on an Apple II. Also shown were a Hires graphic package written in FORTH; A "de-FORTHer" program that takes FORTH words down to their component parts; and a 64 bit quad precision math package. FORTH concepts such as the word DEPTH and .S (a non-destructive stack print out) were also discussed. Meetings are held monthly at THE COMPUTER & THINGS STORE; 3460 S.W. 185th, Suite D; Aloha OR 97006

TULSA COMPUTER SOCIETY: A FORTH Interest Group has been formed in Tulsa, OK under the auspices of the Tulsa Computer Society. The group has 6502 figFORTH running on several Apple II's and 8080 figFORTH running on a Compucolor and a MITS Altair using CP/M and Micropolis Drives. For meeting information contact Art Gorski; c/o The Tulsa Computer Society; P.O. Box 1133; Tulsa, OK 74103 or call (918) 743-0113; (918) 743-4081

SOCIETE D'INFORMATIQUE AMATEUR DU QUEBEC: Has a FORTH group (French!) that meets every other week. Anyone from the Quebec area who would like meeting information is invited to contact Gilles Paillard; 1310 Des Pins Est; Ancienne-Lorette; Quebec, Canada G2E 1G2 or call (418) 871-1960

FIGSA: South Africa has a very active FORTH Interest Group meeting monthly and currently is offering FORTH mini-courses to ground users in the fundamentals. Interested persons in the Johannesburg and Pretoria locales can get more information regarding meetings and courses by contacting Ed Murray; FORTHWITH COMPUTERS; PO Box 27175; Sunnyside Pretoria 0132, South Africa

SOUTHERN CALIFORNIA fig: Attendees numbered approximately thirty-five and most had up and running FORTH systems. Three books were reported: Threaded Interpretive Language by Loeliger, which steps the reader through Z-80 source code of fig-FORTH for the TRS-80; MINT, Machine-Independent Organic Software Tools, by Godfrey, et al.; and FORTH SYSTEM GUIDE by Ting which now has the assembler in its final chapter. The formation of an Organge County fig group was begun.

Martin Tracy of MicroMotion discussed Implementing Strings in FORTH, their 8th chapter in "FORTH-79 Tutorial and Reference Manual" (for the APPLE II). This string package compares, concatenates, converts and arrays with words like GET$, INPUT$ and IN$ (which indexes into the $tring).

AN OPEN RESPONSE

We continually receive letters asking if FORTH can be installed on a particular computer, particularly those without direct access mass storage or an ASCII terminal (i.e. PET, Vip, and Kim). Often, similar queries reflect a desire to use cassette tape. This summary gives the general characteristics of a system in which FORTH will be responsive. For fig-FORTH installation, an assembler is also needed.

FORTH is an interactive, compiled language. This statement may be expanded to conclude that compilation requires mass storage for source text; it must be random access to be interactive. A terminal is also needed, as a hex keypad cannot be deemed interactive. The character set must be complete for program portability, reflecting the commonality of language.

Requirements to execute:

1. A random access mass storage device with direct access to sector read/write i.e. disk or diskette.
2. 16 Kilobytes of ram.
3. A keyboard input with at least the full upper case ASCII character set.
4. A display of at least 64 characters by 16 lines.

Requirements to install:

1. An assembler that can accept about 80K of source producing about 5.5K of object, either memory or disk.

Requirements derived from the FORTH-79 Standard:

1. 2000 bytes of memory for application dictionary (beyond FORTH, stacks and disk buffers).
2. Stacks of 64 and 48 bytes
3. Mass storage of 32 blocks of 1024 bytes
4. An ASCII terminal

If you are missing any of these elements, we express our condolences. You will have to tolerate an irregular installation and suffer portability problems. This curse is not caused by FORTH but by the shortsightedness of hardware vendors. FORTH is an environment in which you can operate as a professional. We know of no professional who would demand to have his terminal line width reduced to 40 characters, have six ASCII characters removed from his keyboard or return his disk to the manufacturer as unnecessary. If FORTH were compromised to less than the above guidelines, we would ultimately be operating from a hex keypad with paper tape.

BENCHMARKING:

Because there is almost universal disagreement on which are the most valid benchmark tests; and because in FORTH memory compactness may be traded off for execution speed at the implementor's option, it is the policy of FORTH DIMENSIONS to minimize the use of benchmark tests that measure speed alone. Such single dimensional tests more precisely measure the speed of a given CPU than the implementation of FORTH itself and encouraging such simplistic testing will probably mean the compactness of FORTH will inevitably suffer. For these reasons FORTH DIMENSIONS is normally only interested in benchmark tests that measure both productivity (useful work) and speed as a better indicator of a given implementations value.

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
(415) 858-2218

Sierra Computer Co.
617 Mark NE
Albuquerque, NM 87123

APPLE

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

6800 & 6809

Talbot Microsystems
5030 Kensington Way
Riverside, CA 92507
(714) 781-0464

TRS-80

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

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

KV33 Corp.
PO Box 27246
Tucson, AZ 85726

68000

Emperical Res. Grp.
PO 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

Elmer W. Fittery
110 Mc Gregor Avenue
Mt. Arlington, NJ 07856
(213) 663-1580

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

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

Henry Laxen
1259 Cornell
Berkeley, CA 94706
(415) 525-8582

John S. James
P. O. Box 348
Berkeley, CA 94701

NEW PRODUCT ANNOUNCEMENT FORMAT

In the interests of comparison uniformity and completeness of data in new product announcements FORTH DIMENSIONS requests that all future new product announcements use the following format:

1. Vendor name (company)
2. Vendor street address (P.O. Boxes alone are not acceptable for mail order)
3. Vendor mailing address (if different from street address)
4. Vendor area code and telephone number
5. Person to contact
6. Product name
7. Brief description of product use/features
8. List of extras included (editor, assembler, data base, games, etc.)
9. List of machines product runs on
10. Memory requirements
11. Number of pages in manual
12. Tell what manual covers
13. Indicate whether or not manual is available for separate purchase
14. If manual is available indicate separate purchase price and whether or not manual price is credited towards later purchase
15. Form product is shipped in (must be diskette or ROM -- no RAM only or tape systems)
16. Approximate number of product shipments to date (product must have active installations as of writing -- no unreleased products)
17. Product Price
18. What price includes (shipping, tax, etc.)
19. Vendor warranties, post sale support, etc.
20. Order turn around time

HELP WANTED

Openings for a project manager and senior programmer. Both positions offer the opportunity to work on a wide variety of projects, including systems programming and real-time scientific and industrial applications. Salary and benefits are excellent. A starting bonus is available for anyone with a substantial FORTH background. Contact FORTH, Inc.; 2309 Pacific Coast Highway; Hermosa Beach, CA 90254; (213) 372-8493

Programmer analyst to work and live in the Miami area who is trained and experienced in the CYBOS language. Contact Keller Industries, Inc., 18000 State Road 9, Miami, FL 33162; (305) 651-7100, ext. 202.

FORTH programmer for computer graphics. Contact Cornerstone Associates, 479 Winter St., Waltham, MA 02154; (617) 890-3773.

RENEW TODAY!

FORTH, INC. NEWS PAGE

This is the first in a series of columns highlighting various activities at FORTH, Inc.

RECENT APPLICATIONS:

In December Chuck Moore completed work on a 24-channel video mixer for Homer & Associates, a producer of films for promotioal and entertainment purposes in Hollywood. This Z-80 based system controls 16 slide projectors, four movie projectors and audio tape. It has mastering and sequencing capabilities which Peter Conn, Homer's president, says are unique in the industry.

In early January American Airlines performed the final acceptance of the LAX outbound baggage system developed by Dean Sanderson and Mike LaManna. The system runs on two PDP-11 computers (one functions as a backup), and controls several conveyor belts, bag encoder stations, electric eye sensors, and printers. It is more accurate and has 25% better performance than the all-assembly language system it replaced.

NEW PRODUCTS:

EXORset polyFORTH pF6809/30, developed by Mike LaManna, is our newest product and runs on the Motorola EXORset 30 -- a microcomputer featuring a 6809 processor, graphics CRT and two mini-floppies in a single compact box. EXORset polyFORTH sells for $4750 and incudes a secial screen editor; a high-speed graphics option with software vector and character generation; labeled graphs with several plotting modes; a "strip-chart" function with snap-shot capabilities, and several demonstration routines. EXORset polyFORTH sells for $4750.00. The option package sells for $500.00. Both will be featured at FORTH, Inc.'s spring seminar series.

FORTH - 79:

Al Krever is working on a new release of polyFORTH scheduled for March. This new release will feature many improvements in all systems, plus greater compatability with the FORTH - 79 Standard.

The FORTH - 79 edition of USING FORTH has been sent to the printers and will be available after mid-February.

POLYFORTH COURSES:

FORTH, Inc. offers two courses-- an introductory course for programmers unfamiliar with polyFORTH and an advanced course designed for those with considerable FORTH experience who desire greater familiarity with system level functions, target compiling and other advanced techniques.

FORTH, Inc.'s course schedule for the next few months is:

Month	Introductory	Advanced
April	6 - 10	13 - 17
May	11 - 15 (tentative)	

Contact Carol Ritscher at FORTH, Inc. (213) 372-8493 for more information.

SEMINARS & WORKSHOPS:

A series of completely new half-day seminars and one-day workshops has been scheduled in several cities. Both present an overview of the features and benefits of polyFORTH for professional users. The EXORset and tis new graphics package will be featured.

City	Seminar	Workshop
Washington, DC	3/19	3/20
Houston	4/21	4/22
Boston	4/23	4/24

Contact Carol Ritscher at FORTH, Inc. (213) 372-8493 for more information.

How to form a FIG Chapter:

1. You decide on a time and place for the first meeting in your area. (Allow about 8 weeks for steps 2 and 3.)
2. Send to FIG in San Carlos, CA 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 Saturday	FIG Monthly Meeting, 1:00 p.m., at Southland Shopping Ctr., Hayward, CA. FORML Workshop at 10:00 a.m.
Southern California 4th Saturday	FIG Meeting, 11:00 a.m. Allstate Savings, 8800 So. Sepulveda, L.A. Call Phillip Wasson, (213) 649-1428.
Massachusetts 3rd Wednesday	MMSFORTH Users Group, 7:00 p.m., Cochituate, MA. Call Dick Miller at (617) 653-6136 for site.
San Diego Thursdays	FIG Meeting, 12:00 noon. Call Guy Kelly at (714) 268-3100 x 4784 for site.

Seattle	Chuck Pliske or Dwight Vandenburg at (206) 542-8370.
Potomac	Paul van der Eijk at (703) 354-7443 or Joel Shprentz at (703) 437-9218.
Tulsa	Art Gorski at (918) 743-0113
Texas	Jeff Lewis at (713) 729-3320 or John Earls at (214) 661-2928 or Dwayne Gustaus at (817) 387-6976. John Hastings (512) 835-1918
Phoenix	Peter Bates at (602) 996-8398
Oregon	Ed Krammerer at (503) 644-2688.
New York	Tom Jung at (212) 746-4062.
Detroit	Dean Vieau at (313) 493-5105.
England	FORTH Interest Group, c/o 38, Worsley Road, Frimley, Camberley, Surrey, GU16 5AU, England.
Japan	Mr. Okada, President, ASR Corp. Int'l, 3-15-8, Nishi-Shimbashi Manato-ku, Tokyo, Japan.
Quebec, Canada	Gilles Paillard at (418) 871-1960.

Publishers Note:

Please send notes (and reports) about your meetings.

RENEW

RENEW TODAY!