# FORTH Dimensions

Volume 6, Number 4

November/December 1984

$2.50

## Forth P-Code Interpreter

## Recursion

## Forth Semaphores

## Run '79 Code on Forth-83

## ANDIF and ANDWHILE

**FORTH Dimensions**
Published by the
Forth Interest Group

**Volume VI, Number 4**
November/December 1984

Editor
Marlin Ouverson

Production
Jane A. McKean et al.

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

Subscription to *Forth Dimensions* is free with membership in the Forth Interest Group at $15.00 per year ($27.00 foreign air). For membership, change of address and/or to submit material for publication, the address is: Forth Interest Group, P.O. Box 1105, San Carlos, California 94070.

## Symbol Table

Simple; introductory tutorials and simple applications of Forth.

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

Advanced; requiring study and a thorough understanding of Forth.

83 — Code and examples conform to Forth-83 standard.

79 — Code and examples conform to Forth-79 standard.

fig — Code and examples conform to fig-FORTH.

FORML — Deals with new proposals and modifications to standard Forth systems.

# FORTH Dimensions

## FEATURES



## DEPARTMENTS

# Letters to the Editor

### Grounded in Data Transfer, and CREATE for Jupiter

Dear Mr. Ouverson:

I would like to comment on "Simple Data Transfer Protocol" by Ericson and Feucht (*Forth Dimensions* VI/2). Figure one showed pin 1 as ground. RS232 designates pin 1 as the chassis ground, while pin 7 is the signal ground. In some computer systems these two grounds may be electrically connected, but in others they are not. Therefore, it is good practice to use pin 7 instead of pin 1 as ground for communications cabling. Figure two showed a loop connection of the control signals on pins 4, 5 and 8. This will work for many systems, but should not be considered universal. The control signals required vary from system to system. Some need pin 6 (data set ready) asserted to enable receiving. Others need no control signals at all.

When working with RS232 ports on various computer systems, I have found it very useful to use a cable matcher. This is a small box with RS232 connectors on both ends and jumpers between the connectors. A cable matcher enables me to test the RS232 port with different control signal loops, as well as with pins 2 and 3 crossed or uncrossed (with or without null modem). I have found that I can get two RS232 ports communicating by trial and error faster than I can by trying to decipher any documentation for the ports.

Also, I am the owner of a Jupiter Ace computer and would like to share some code with other Jupiter programmers. I am disappointed by the Jupiter's **DEFINER DOES>** pair, which takes the place of **CREATE DOES>**. For simple defining words, they work fine. However, constructing a defining word that constructs defining words, as presented in Henry Laxen's fine articles (*Forth Dimensions* IV/2,3), is beyond the capabilities of **DEFINER DOES>**. Redefining **DOES>** as in figure one will allow **CREATE DOES>** to be used as by the Forth-79 Standard. I used Glen Haydon's book, *All About Forth*, as a reference to aid in the development of the definitions.

This demonstrates that, although the Jupiter does not contain a complete Forth-79 implementation, alterations to the system to make it more closely conform are quite easy. Ease of system alteration is one of the outstanding characteristics of Forth.

Sincerely,

Ed Schmauch
Conoco, Inc.
P.O. Box 1267
Ponca City, Oklahoma 74603

```
  16 BASE C!
  FF0 CONSTANT   DODOES
: COMPILE  R> DUP  @   ,   2+ >R  ;
: <;CODE> R> CURRENT @   @ 1+   !  ;
: DOES> COMPILE <;CODE>  CD  C,  DODOES  ,  ; IMMEDIATE
  DECIMAL
```

**Figure One**

### Coding for Dollars, and Wanted: Slow Editors

Dear FIG:

Your recent articles on "PL/I Data Structures" (*Forth Dimensions* V/6) and "Procedural Arguments" (VI/2) are the wave of the future—at least of Forth's future. If Forth is to be more than a process control language, it must live up to Moore's claim (in these pages) that Forth can do anything any other language can do, only more elegantly.

To that end, I would like to suggest a competition organized by FIG, to be held in these pages, in which 1) the major features of all major languages are defined by an expert committee, and 2) annual prizes are given for those published articles which best describe how these features can be implemented in Forth.

Prizes should be awarded for 1) the most complete implementation, 2) the most intelligible implementation, 3) the simplest implementation, 4) the most elegant (i.e. combination of all the above) implementation.

Prizes should consist of a free year's membership in FIG. Furthermore, as each of the major languages (I nominate COBOL, RPG-II, PL/I, Pascal, Modula-II, Ada, C, Fortran, APL, Lisp and Prolog) is completed, articles relevant to it should be collected into monograph form and authors of those articles should be given a copy of that monograph.

The real winner in this competition would be the computing community, which would gain the ability to use the best of each language in a way uniquely suited to the purpose at hand. If this suggestion is taken as seriously as I hope it will be, I would like as my reward for suggesting it a standing invitation to have the pleasure of the company of Henry Laxen and Bill Ragsdale for lunch or dinner, which I shall gladly buy. The opportunity to be surrounded by their kind of brilliance (their columns are worth the entire price of admission) could be the prize for the year's best article.

Finally, an editorial suggestion. You need someone as slow to learn as I on your editorial board. The standing joke in our local FIG chapter meetings is my

# Editorial

# Points of Departure

As we put this issue together, the FORML tour group is in the midst of last-minute preparations for its trip to Taiwan, Hong Kong and China. All are looking forward to the technical interaction with FIG members, computer professionals and academicians in those countries, although I've heard rumor that at least one traveler will forego one of the conferences for the sake of cultural exchange (could it be shopping?). Barring terminal jet lag, you'll read about the conferences—and maybe even the shopping—in an upcoming issue.

If you are one of those who plans to stay in *terra cognita* this year, I hope you at least treat yourself to the Forth convention and to the FORML conference, both in November. The programs for both events promise to deliver double doses of both conventional and innovative Forth wisdom. Although it will be no substitute for being there, as with the journey to the East, we will report as many of the items of interest as these pages will allow.

Meanwhile, back to the issue at hand. We are happy to present you with an index to the last volume of *Forth Dimensions*. Write to let us know if you find it useful and would like to see other volumes indexed in the same way.

The feature which stands out the most, perhaps, is Al Monroe's p-code interpreter written in Forth. As he explains, it is intended to be both useful and educational. We feel it is particularly appropriate for Pascal programmers to use as a point of departure into the world of Forth. As a note of explanation to you style purists, it is intentionally written in a way to show how Pascal code can be mapped onto Forth. As an interesting exercise, it coincides with a reader's request in this issue's "Letters to the Editor."

We continue looking for simple applications to publish in these pages. There are few better ways to appreciate Forth than by study of a clear example of working code alongside a lucid explanation with just the right amount of detail. We have received some promising contributions and look forward to receiving many more. It's always good to hear from the FIG membership, so keep those letters and articles coming!

—*Marlin Ouverson*
*Editor*

---

ratings of your articles: each receives a number equal to the number of times I had to read it before I understood it. The PL/I article, which I give a 10 of 10 for insight, also got a 10 for the number of times I read it before I understood what was going on. I'm at 4 for the equally insightful procedural arguments article, and counting.

Sincerely yours,

Henry J. Fay
4020 East Road
Cazenovia, New York 13035

**Mixed INTEGER Review;**
**Consistency Constituent**

Dear Sir:

I read with interest "The Integer Solution" by Marc Perkel (*Forth Dimensions* VI/2). Since it was tagged with a FORML label, I felt a discussion of the ideas presented was in order. First, his idea for two code fields is interesting and possibly useful in areas other than **INTEGER**s. However, his examples and the idea of an **INTEGER** touch directly at the core philosophy behind Forth.

Forth uses postfix notation for most of its syntax, with the exception of **'**, **FIND** and defining words. Assigning a value to an **INTEGER** is done using prefix notation and would be used extensively. This would be confusing. Is Forth to be a consistent language, or are we to have conflicting rules? Do we want to have another English (i before e except in receipt and a few other places)?

Mr. Perkel says that he eliminates @ and **!**. He does not. He eliminates @ and replaces **!** with ->. The gain in brevity is only half his claim. Thus, I feel **INTEGER** is not a useful addition to the Forth standard because the loss in consistency is not offset by the slight gain in source code brevity.

Another complaint I have is that Mr. Perkel's examples are not a comparison between **INTEGER** and **VARIABLE** but between using variables for storage and using the stack for storage. The example definition for **BOX** could be written using variable storage, and to me would be just as readable, even with the addition of the @ after the variable name. The definition of **BOX** using the stack will be harder to read and understand, as will most any other word defined to use the stack for data storage. The advantage of using the stack is not in having readable code, but in having "reentrant" code. Unless a solution uses recursion, reentrant code is not needed for most application programs.

The code in Marc's first figure is not an application but a system operation. In it, he assumes a system variable (**BLK**) has been redefined as an **INTEGER**. This is a very, very bad idea. Systems words must be reentrant if Forth is to be used in a multi-tasking or multi-user environment. While Mr. Perkel's system may be single user, and he may have no plans to do multi-tasking, any Forth system has the potential for multi-tasking. I sincerely believe that this is a strength of the original design of fig-FORTH and is not something that should be left out of future language definitions.

The standard definition of **BLK** is as a **USER** variable. It contains another level of indirection via the UP (user pointer) that makes it possible for each task or user to have a complete set of system variables. This is done by the operating system when switching tasks and is transparent to the user. With Mr. Perkel's **INTEGER BLK**, separate users trying to access the disk at the same time would end up getting the same data, that contained in the second user's **BLK**. His definition for **MORE** is easier to read and understand, I just think the standard **BLK** (with associated @) should be used instead.

A question that I have for the standards committee concerns defining words. Are they consistent with postfix notation? At first glance, it appears they are not consistent. However, :, **VARIABLE**, etc. do not get the name from the data stack but rather from the input stream. This was one of the conceptual problems I encountered when learning Forth several years ago. Now the order : **newname** seems natural to me, but should the language definition be changed to eliminate an inconsistency in the syntax? This could easily be done with a change to **INTERPRET** incorporating a check for a defining word after determining a token is not in the dictionary and is not a number. This solution does not allow for defining 5 as a constant with the name 5, however, and so has problems of its own. I am for keeping the syntax of Forth constant with postfix notation but I am not clear in my own mind that the defining words really constitute a problem. What do other people (and especially newcomers to Forth) think?

Sincerely,

Dr. Ken Butterfield
2020 - 23rd Street, Apt. C
Los Alamos, New Mexico 87544

*Editor's note: Your observation that Forth's defining words do not appear consistent with postfix notation is a correct one. However, no proposed change has recevied the required accolades; detailed discussion will fetch up problems of state-smartness, string stacks, bit switches and other sleeping dogs, to say nothing of the functionality and inertia of the present syntax.*

### Search for Model III Source

Dear FIG:

I'm looking for a fig-FORTH or 79-Standard system on disk for my TRS-80 Model III. Not the CP/M version, but public-domain software with source code. I'm operating under MMS-FORTH but much of the kernel is not with source code and I can't sell the system with my own programs.

I am a FIG member who needs direction. Thank you.

Arthur Wendover
Box 263
Isafjordur, Iceland

P.S. Your Volume VI, Number 1 issue is very useful and interesting, especially the list handling article.

# Forth P-Code Interpreter

*A.J. Monroe*
*Los Angeles, California*

In a series of three articles in *BYTE* magazine (September, October, November 1978), Kin-Man Chung and Herbert Yuen published a "Tiny" Pascal compiler (written in North Star BASIC), a p-code-to-8080 translator (in the same language) and a p-code interpreter written in "Tiny" Pascal.

The p-code generated by the compiler is relocatable and completely transportable, whereas the output of the translator is unique to the Intel 8080 microprocessor instruction set (or Intel 8085 or Zilog Z-80). Further, since the published interpreter is written in "Tiny" Pascal, it can only be utilized with the published translator on an 8080-compatible computer.

It recently occurred to this writer that a p-code interpreter could be easily written in Forth. This would serve two purposes.

First, Forth is currently available on a wide selection of microprocessor types at a very reasonable price, through the good offices of the Forth Interest Group, among others. Such an interpreter would, of course, execute considerably slower than the translated p-code, but this is offset by the fact that this approach effectively circumvents the lack of an appropriate translator.

Second, if the interpreter were to be written closely following the Chung/Yuen interpreter, it could serve as a unique way of introducing those already familiar with Pascal to the Forth language. Personally, I have always found it easier to learn a language from the study of examples: one example in a language with which I am already familiar and another example being the same program in the new language.

I do not mean to imply that the reader will find this article to be a tutorial on Forth. Anyone who is totally unfamiliar with the language will have to do considerable boning up to fully understand the Forth listing; but some explanation will be given as we go along, and as a result the reader should hope to gain some appreciation of the similarities and differences between Forth and Pascal—and incidentally gain a program of interim usefulness. In particular, for those who already understand Pascal, I will wager that the Forth version of the interpreter will be surprisingly understandable.

The caveat "interim usefulness" deserves some elaboration. The only strong arguments that this writer has ever heard against interpreters are that they are "slow compared to compiled code" and that they tend to be "memory hogs" in the sense that the interpreter and the program to be interpreted must reside in memory simultaneously—to the detriment of available programming space. Interpreters are not small programs, e.g. the Forth interpreter is 5800+ bytes and, including this writer's version of Forth, requires very nearly 16K of memory.

In the present instance, the first objection is perhaps the more serious one. Consider the Pascal listing in figure one. This program writes and reads to absolute memory the ASCII characters from A to the left bracket symbol and outputs them to the console. The program generates thirty-one p-codes (a total of 124 bytes). When interpreted, the program requires the execution of 555 instructions because of the FOR loop construct. If the program is translated to 8080 object code (again 124 bytes) and executed, it will complete execution in somewhat under one second. If the p-code is interpreted by the object code version of the Chung/Yuen interpreter, execution will be completed in about five seconds, i.e. five times slower than the execution time of the translated program. If this same p-code is interpreted by the interpreter written in Forth, execution will require about twenty-three seconds, another factor of five in increased execution time.

This last execution time is not a serious objection if the Forth interpreter is being used for debugging purposes, but clearly it is not likely to be acceptable after debugging is completed, especially when you know that you can get twenty-three times faster execution from the object code! As noted above, the program is intended primarily as a cross-programming example, an aid to understanding Forth given that the reader is already familiar with Pascal.

**Forth and Pascal: Some Comparisons**

Figure two lists the Chung/Yuen interpreter written in Pascal. Figure three lists a Forth version which, in terms of structure, emulates closely the listing in figure two. The chief difference is one of syntax and mnemonics. But the casual reader who is already familiar with Pascal should see many points of similarity between the two listings.

```
VAR I : INTEGER;
MN : ARRAY[26] OF INTEGER;
BEGIN
     FOR I:=0 TO 26 DO
        BEGIN MEM[I]:=I+65; MN[I]:=MEM[I];
              WRITE(MN[I])
        END;
     WRITE(10,13)
END.
```

**Figure One**
**"Tiny" Pascal program that reads and writes to absolute memory locations**

This congruence is not entirely an artifice designed by the writer. There are, it is true, many significant differences in mnemonics and syntax between the two languages, but consider the similarities:

Both languages require that variables and constants be declared before use. In Pascal this is done right up front; in Forth any old place in the program will suffice, so long as it's before first usage. Both languages are highly structured and use similar constructs, such as IF...THEN...ELSE... and BEGIN... END. The use of GOTO is impossible in Forth and, although not forbidden in Pascal, it is frowned upon by the purist and is not supported in "Tiny" Pascal in any event.

Rudimentary Forth does not support a **CASE** statement, nor does it have **ARRAY**. But Forth is inherently extensible (as opposed to Pascal and most other languages) and such constructs may, therefore, be added quite easily. This is illustrated in the listing of figure three in screens 54, 55 and 56.

Pascal supports "procedures" and "functions." Entirely analogous to the Pascal procedure, in Forth we have the "word" (colon) construct. In fact, *everything* in Forth is a procedure, including the main program, which is simply one more procedure which invokes all the others as needed and is itself just one more word in the language. Pascal is very similar in this respect.

On the other hand, Forth uses postfix (reverse Polish) notation, whereas Pascal and most other languages utilize infix notation. This is usually the biggest stumbling block to understanding encountered by the newcomer to the language—unless, of course, he cut his teeth on an HP calculator.

Invoking the name of a variable in Forth ordinarily puts the address of that variable on the stack, whereas in Pascal the current value of the variable is placed on the stack. But, if we wish, we can alter (extend) Forth to do the same as Pascal via the **TO-VARIABLE** construct shown in screen 57.

Both languages are stack-oriented (zero address) languages.

```
/P-CODE INTERPRETER BY H.YUEN/
CONST U=15;BPLIM=5;SIZE=500;SIZE1=480;
VAR Z,P,B,T,BP,P0,TP,CMD,I,J,K,STOP:INTEGER;
    S:ARRAY[SIZE] OF INTEGER;
    TRACE:ARRAY[U] OF INTEGER;
    MN:ARRAY[26] OF INTEGER;
    BREAK:ARRAY[BPLIM] OF INTEGER;
/IMPORTANT GLOBAL VARIABLES:
 P:PROGRAM COUNTER                B:BASE POINTER
 T:STACK POINTER                 BP:BREAK PNT INDX
TP:TRACE STACK PNTR               K:INSTRUCTION COUNTER
 S:DATA STACK                     Z:STRT ADDR OF P-CODE /
FUNC BASE(LEV);
 VAR B1:INTEGER;
 BEGIN B1:=B;
  WHILE LEV>0 DO BEGIN
   B1:=S[B1];LEV:=LEV-1 END;
  BASE:=B1
 END /BASE/;
PROC INIT;
 VAR I:INTEGER;
 BEGIN T:=0;B:=1;P:=0;STOP:=0;
  S[1]:=0;S[2]:=0;S[3]:=-1;
  P0:=0;TP:=U;K:=0;
  FOR I:=0 TO U DO TRACE[I]:=-1
 END /INIT/;
PROC CRLF;
 BEGIN WRITE (10,13) END;
PROC EXEC;
 VAR X,A,L,F,IDX:INTEGER;
 BEGIN X:=P SHL 2+Z;
  A:=MEM[X+3] SHL 8 +MEM[X+2];
  TP:=TP+1;IF TP>U THEN TP:=0;
  TRACE[TP]:=P;
  P:=P+1;P0:=P;K:=K+1;
  F:=MEM[X];
  IF F<=8 THEN IDX:=0
     ELSE BEGIN IDX:=1;F:=F-16 END;
  CASE F OF
0:BEGIN T:=T+1;S[T]:=A END;
1: CASE A OF
0:BEGIN /RETURN/
   T:=B-1;B:=S[T+2];P:=S[T+3] END;
   1 :S[T]:=-S[T];
    2 :BEGIN T:=T-1;S[T]:=S[T]+S[T+1] END;
```

**Figure Two**
**P-code interpreter written in "Tiny" Pascal**

On the other hand, there are profound differences between the two languages. A Pascal program must first be compiled to be executed. To the contrary, the Forth listing in figure three is executable as soon as it has been typed into Forth, simply by invoking the procedure **MAIN** of screen 67 by typing its name.

Forth supports an "immediate" mode of execution as is usual with interpreters, and also a "compile" mode. Pascal is usually compiled, and this writer is unaware of any implementation which permits an immediate mode of execution.

Forth supports a native code assembler so that "code" words can be generated and used as simply and naturally as words defined by the colon construct. Three crucial examples of this feature are shown in screen 56. Such linkage is

```
SCR # 54
( DR. EAKER'S CASE CONSTRUCT WITH A SLIGHT MODIFICATION )
( SEE FORTH DIMENSIONS VOL 2 NO. 3 PG 37-40 )
: CASE ?COMP CSP @ !CSP 4 ; IMMEDIATE
: (OF) OVER = IF DROP 1 ELSE 0 ENDIF ;
: OF 4 ?PAIRS COMPILE (OF) COMPILE 0BRANCH
    HERE 0 , 5 ; IMMEDIATE
: ENDOF 5 ?PAIRS COMPILE BRANCH HERE 0 , SWAP 2
        [COMPILE] ENDIF 4 ; IMMEDIATE
: ENDCASE 4 ?PAIRS COMPILE DROP BEGIN SP@ CSP @ = 0 =
          WHILE 2 [COMPILE] ENDIF REPEAT CSP ! ; IMMEDIATE

: 1- 1 - ;



-->

SCR # 55
( <OF, >OF, AND RNG-OF EXTENSIONS TO DR. EAKER'S CASE )
: (<OF) OVER > IF DROP 1 ELSE 0 ENDIF ;
: <OF 4 ?PAIRS COMPILE (<OF) COMPILE 0BRANCH
      HERE 0 , 5 ; IMMEDIATE
: (>OF) OVER < IF DROP 1 ELSE 0 ENDIF ;
: >OF 4 ?PAIRS COMPILE (>OF) COMPILE 0BRANCH
      HERE 0 , 5 ; IMMEDIATE
: RANGE >R OVER DUP R> 1+ < IF SWAP 1- > IF DROP 1 ELSE 0
          ENDIF ELSE DROP DROP 0 ENDIF ;
: RNG-OF 4 ?PAIRS COMPILE RANGE COMPILE 0BRANCH HERE 0 , 5 ;
IMMEDIATE




-->

SCR # 56
( CODE WORDS LSHFT AND SHL )
CODE LSHFT                    ( LOGICAL SHIFT LEFT ONE BIT )
     H POP                    ( GET DATA WORD FROM STACK )
     A XRA                    ( CLEAR THE CARRY )
     L A MOV                  ( GET LEAST SIG. BYTE )
     RAL                      ( SHFT LEFT INTO CARRY )
     A L MOV                  ( RSTR LEAST SIG. BYTE )
     H A MOV                  ( GET MOST SIG. BYTE )
     RAL                      ( SHFT LEFT - CY TO LSB )
     A H MOV                  ( RSTR MOST SIG. BYTE )
     H PUSH                   ( PUT WORD BACK ON STACK )
     NEXT JMP                 ( RETURN TO FORTH )

: SHL 0 DO LSHFT LOOP ;

-->
```

**Figure Three**
**Forth p-code interpreter**

non-existent in Pascal and the majority of other high-level languages. The SHR, SHL and CALL reserved words of "Tiny" Pascal had to be built into the compiler, a non-trivial task at best. They are simply and directly mechanized as code words in Forth, as is illustrated in screen 56.

But it is not the differences between Pascal and Forth that were of significance in developing the listing shown in figure three. Rather, it is the fact that both languages are sufficiently similar in construction that the listing in figure two could be translated with almost one-to-one correspondence into Forth constructs, and almost as rapidly as it was possible to type! The writer must confess that this high degree of correspondence was not at all self evident at the outset. In retrospect, this appears to have been largely due to the superficial differences in mnemonics.

### The Forth Interpreter

The Pascal p-code interpreter makes extensive use of PROC, CASE and ARRAY. PROC, short for "procedure," presents no problems in direct translation to Forth. As noted earlier, Forth's colon construct is completely analogous to the Pascal PROC. However, rudimentary Forth does not support CASE or ARRAY, and they must be added to the language if a direct emulation of the interpreter is to be achieved.

A **CASE** construct is shown in screen 54 and 55. This particular construct was developed by Dr. Eaker and is explained in detail in *Forth Dimensions* (II/3). The reader should refer to that excellent article for details. Screen 55 is this writer's augmentation of Dr. Eaker's **CASE** construct. The **<OF** tests for "less than," **>OF** tests for "greater than" and **RNG-OF** tests for inclusion in the range between two integers. These three additions should be easy to understand from the explanations given in Dr. Eaker's article on the earlier **CASE** constructs.

There are several ways in which an **ARRAY** construct may be implemented in Forth. Screen 59 illustrates one such definition. The word **ARRAY** is used as follows:

size **ARRAY** array-name

During compile time, size reserves that number of sixteen-bit words with index addresses 0 through size-1. At execution time, invoking array-name will cause the top number on the data stack to be interpreted as the index address of the word (array element) to be accessed in the array. This number is first checked to see that it is within the valid index address range. If not, an error message is output to the console and program execution is terminated. Otherwise, the absolute memory address of the desired array element replaces the index address on the top of the data stack. To store an item, one types:

value-to-be-stored index-address array-name !

To retrieve an element from the array, one types:

```
SCR # 57
< CODE WORD RSHFT, SHR, AND TO-VAR >
CODE RSHFT  H POP  A XRA  H A MOV  RAR  A H MOV
     L A MOV  RAR  A L MOV  H PUSH  NEXT JMP

: SHR 0 DO RSHFT LOOP ;

< DEFINITION OF BARTHOLDI'S TO-VAR >
< SEE FORTH DIMENSIONS VOL 1 NO. 4 PG 38-40 >
0 VARIABLE %TO           : TO 1 %TO ! ;

: TO-VAR <BUILDS HERE 2 ALLOT !
         DOES> %TO @ IF ! 0 %TO ! ELSE @ ENDIF ;
-->
```

```
SCR # 58
< CODE WORD CALL >
CODE  CALL  < ABSOLUTE MEMORY CALL TO OBJ CODE ROUTINE >
  < SAVE THE FORTH INSTR. PNTR IN THE DICTIONARY >
  I' L MOV  I H MOV  HERE 6 + SHLD
  HERE 5 + JMP < JUMP OVER THE STORE LOCATION >
  0 A MVI < THIS IS STORE LOCATION SAVED BY DUMMY INSTR. >
  H POP   < GET THE ADDRESS TO BE CALLED >
  HERE 4 + SHLD < PUT ADDR INTO CALL INSTR. >
  0 CALL  < DUMMY CALL FILLED BY ABOVE >
  < OBJ LNG RTN WILL RETURN HERE >
  < NOW RSTR FORTH INSTR. PNTR >
  HERE 9 - LHLD  H PUSH  I POP
  NEXT JMP  < RETURN TO FORTH >
-->
```

```
SCR # 59
< DEFINITION OF ARRAY >
: ERRA ." ARRAY INDEX ERROR " . CR @ 1- 0
       ." ARRAY INDEX RANGE = " . ."    " . CR QUIT ;
: ARRAY <BUILDS DUP , < DUP SIZE & SAVE IN PARAM FIELD >
         2 * < # OF BYTES TO ALLOT >
         HERE < CURRENT DP >
         2DUP + 2+ < UPPER LIMIT OF DO LOOP >
         SWAP DO 0 I ! 2 +LOOP < CLEAR THE ARRAY >
         ALLOT < RESERVE DICTIONARY SPACE >
        DOES> 2DUP @  < < TEST FOR UPPER ARRAY LIMIT >
        IF SWAP DUP -1 > < TEST FOR LOWER LIMIT >
        IF 2 * + 2+ < SET ARRAY ADDR ON STACK >
        ELSE ERRA ENDIF ELSE SWAP ERRA ENDIF ;
-->
```

```
SCR # 60
< CONSTANTS, TO-VAR, AND MN FOR PASCAL INTERPRETER >
15 CONSTANT U      5 CONSTANT BPLIM    500 CONSTANT SIZE
480 CONSTANT SIZE1
0 TO-VAR Z     0 TO-VAR B1    0 TO-VAR P    0 TO-VAR B
0 TO-VAR LEV   0 TO-VAR T     0 TO-VAR BP   0 TO-VAR BASER
0 TO-VAR P0    0 TO-VAR TP    0 TO-VAR CMD 0 TO-VAR X
0 TO-VAR J     0 TO-VAR K     0 TO-VAR STOP   0 TO-VAR PC
0 TO-VAR IDX 0 TO-VAR EXIT    0 TO-VAR A   0 TO-VAR F
SIZE ARRAY S     U ARRAY TRACE      27 ARRAY MN
BPLIM ARRAY BREAK    0 TO-VAR L   0 TO-VAR NN
76 0 MN ! 73 1 MN ! 84 2 MN ! 79 3 MN ! 80 4 MN !
82 5 MN ! 76 6 MN ! 79 7 MN ! 68 8 MN ! 83 9 MN !
84 10 MN ! 79 11 MN ! 67 12 MN ! 65 13 MN ! 76 14 MN !
73 15 MN ! 78 16 MN ! 84 17 MN ! 74 18 MN ! 77 19 MN !
80 20 MN ! 74 21 MN ! 80 22 MN ! 67 23 MN ! 67 24 MN !
83 25 MN ! 80 26 MN !   -->
```

```
SCR # 61
( TERMINAL INPUT FOR PASCAL INTERPRETER )
: ERRD CR ." SYNTAX ERROR " CR DROP ;  0 TO-VAR SGN
: RDCHR BEGIN 35 P@ 2 AND UNTIL 34 P@ 127 AND ;
: WRCHR BEGIN 35 P@ 1 AND UNTIL 34 P! ;
: RDDEC 0 DUP DUP TO EXIT TO SGN BEGIN RDCHR DUP WRCHR DUP
        CASE      13 OF 1 TO EXIT ENDOF
                48  57 RNG-OF 48 - SWAP 10 * + ENDOF
                    45 OF SGN 0 = IF TO SGN ELSE ERRD ENDIF
             ENDOF ERRD ENDCASE EXIT UNTIL DROP CR
       SGN 0 > IF MINUS ENDIF ;
: RDHEX 0 DUP TO EXIT BEGIN RDCHR DUP WRCHR DUP
       CASE      13 OF 1 TO EXIT ENDOF
              48  57 RNG-OF 48 - SWAP 16 * + ENDOF
              65  70 RNG-OF 55 - SWAP 16 * + ENDOF
              ERRD
       ENDCASE EXIT UNTIL DROP CR ;   -->


SCR # 62
( SBASE AND INIT FOR PASCAL INTERPRETER )
: SBASE B TO B1 BEGIN LEV WHILE B1 S @ TO B1
                                   LEV 1- TO LEV
                                 REPEAT
        B1 TO BASER ;
: INIT 0 TO T 0 TO P 1 TO B 0 TO STOP 0 1 S ! 0 2 S !
       -1 3 S ! 0 TO P0 U TO TP 0 TO K U  0
       DO -1 I TRACE ! LOOP ;

: <= 1+ < ;   : >= 1- > ;   : <> = IF 0 ELSE 1 ENDIF ;
: T-1 T 1- TO T ;   : T+1 T 1+ TO T ;
: NOT 0= IF 1 ELSE 0 ENDIF ;


-->


SCR # 63
( BEGINNING OF EXEC FOR PASCAL INTERPRETER )
  : S(T) T S @ ;   : S(T+1) T 1+ S @ ;
: S(T)= T S ! ;
: EXEC P 2 SHL Z + TO X  X 3 + C@ 8 SHL
      X 2+ C@ + TO A   TP 1+ TO TP   TP U 1- > IF 0 TO TP ENDIF
      P TP TRACE !   P 1+ DUP TO P TO P0    K 1+ TO K
      X C@ TO F F  8 <= IF 0 TO IDX ELSE 1 TO IDX F 16 - TO F
      ENDIF  F
       CASE  0  OF T+1 A S(T)= ENDOF
             1  OF A
                CASE 0 OF B 1- TO T T 2+ S @ TO B
                          T 3 + S @ TO P                 ENDOF
                  1 OF S(T) MINUS S(T)=                ENDOF
                  2 OF T-1 S(T) S(T+1) + S(T)=         ENDOF
                  3 OF T-1 S(T) S(T+1) - S(T)=         ENDOF
                  4 OF T-1 S(T) S(T+1) * S(T)=         ENDOF -->


SCR # 64
( EXEC CONTINUED )
                 5 OF T-1 S(T) S(T+1) / S(T)=               ENDOF
                 6 OF S(T) 1 AND S(T)=                      ENDOF
                 7 OF T-1 S(T) S(T+1) MOD S(T)=             ENDOF
                 8 OF T-1 S(T) S(T+1) = S(T)=               ENDOF
                 9 OF T-1 S(T) S(T+1) <> S(T)=              ENDOF
                10 OF T-1 S(T) S(T+1) < S(T)=               ENDOF
                11 OF T-1 S(T) S(T+1) >= S(T)=              ENDOF
                12 OF T-1 S(T) S(T+1) > S(T)=               ENDOF
                13 OF T-1 S(T) S(T+1) <= S(T)=              ENDOF
              14 OF T-1 S(T) S(T+1) OR S(T)=               ENDOF
               15 OF T-1 S(T) S(T+1) AND S(T)= ENDOF
              16 OF S(T) NOT S(T)=                         ENDOF
              17 OF T-1 S(T) S(T+1) SHL S(T)=              ENDOF
              18 OF T-1 S(T) S(T+1) SHR S(T)=              ENDOF
              19 OF S(T) 1+ S(T)=                     ENDOF -->
```

index-address array-name @

In order to make the one-to-one correspondence between figures two and three more evident, we have defined the words **S(T)**, **S(T+1)** and **S(T)=** on screen 63. **S(T)** retrieves the Tth array element of array S and places it upon the top of the data stack. Similarly, **S(T+1)** retrieves the T+1st array element and **S(T)=** stores the data word from the top of the stack into the Tth element. For example, the code sequence

**S(T) S(T+1) + S(T)=**

sets the Tth element of array S to the sum of itself and the T+1st element.

As mentioned earlier, Bartholdi's **TO** construct (screen 57) is used to place the value of a variable on the stack when its name is invoked. To store a value into the variable, one types:

value **TO** variable-name

For example, A **TO** P puts the value of variable A on top of the stack and then stores it into variable P.

In the Pascal listing in figure two, array MN is used to store the mnemonics of the opcodes which are read from memory. In the Forth listing, these mnemonics are simply stored directly into the **MN** array (screen 60). This could, of course, have been done in Pascal as well.

The "Tiny" Pascal reserved words SHR, SHL and CALL are not normally a part of Forth syntax. They can be directly implemented using the Forth **CODE** words shown in screens 56 and 57.

The READ and WRITE constructs of Pascal are emulated in figure three as **RDCHR**, **WRCHR**, **RDDEC** and **RDHEX**. **RDCHR** reads an ASCII character from the console and places it on the stack, and **WRCHR** takes an ASCII character off the stack and writes it to the console. These words are defined in screen 61. In the writer's system, the input/output flag port is 35. If bit one of this port is set, the port is ready to accept an output. If bit two is set, the port has a character ready to be input. The data port is 34.

```
SCR # 65
( EXEC CONTINUED )
                20 OF S(T) 1- S(T)=                            ENDOF
                21 OF T+1 T 1- S @ S(T)=                       ENDOF
                 ."   ILLEGAL OPR" CR 1 TO STOP
                ENDCASE ( OF A )   ENDOF
2 OF X 1+ C@ DUP TO L 255 = IF S(T) C@ S(T)=
                   ELSE IDX IF A S(T) + TO A ENDIF T 1+ IDX -
                        TO T L TO LEV SBASE BASER A + S @ S(T)=
                         ENDIF   ENDOF
3 OF X 1+ C@ DUP TO L 255 = IF S(T) T 1- S @ C! T 2 - TO T
                             ELSE IDX IF T 1- S @ A + TO A ENDIF
                                    L TO LEV SBASE BASER A + S(T)
                                    SWAP S ! T 1- IDX - TO T
                              ENDIF                  ENDOF
4 OF X 1+ C@ DUP TO L 255 = IF S(T) CALL T-1
      ELSE L TO LEV SBASE BASER T 1+ S ! B T 2+ S !  -->


SCR # 66
( EXEC CONTINUED )
     P T 3 + S ! T 1+ TO B A TO P ENDIF    ENDOF
5 OF T SIZE1 A - >
      IF ."   STACK OVERFLOW " CR 1 TO STOP
      ELSE T A + TO T ENDIF ENDOF
6 OF A TO P ENDOF
7 OF X 1+ C@ S(T) =
       IF A TO P ENDIF T-1 ENDOF
8 OF A CASE
       0 OF T+1 RDCHR S(T)= ENDOF
       1 OF S(T) WRCHR T-1 ENDOF
       2 OF T+1 RDDEC S(T)= ENDOF
       3 OF S(T) WRCHR T-1 ENDOF
       4 OF T+1 RDHEX S(T)= ENDOF
       5 OF S(T) .4H T-1 ENDOF -->


SCR # 67
( EXEC CONTINUED )
       8 OF T DUP   S(T) -
            DO I S @ WRCHR LOOP T S(T) - 1- TO T ENDOF
 ."  ILLEGAL CSP " CR 1 TO STOP
ENDCASE ( OF A )   ENDOF
 ."  ILLEGAL OPCODE " CR 1 TO STOP
ENDCASE ( OF F & END OF EXEC ) ;
0 TO-VAR ID
-->


SCR # 68
( PROCEDURES PCODE AND CKBP OF PASCAL INTERPRETER )
: PCODE PC 2 SHL Z + TO X X C@ 3 * DUP TO NN
         24 <= IF 32 TO ID ELSE NN 48 - TO NN
        88 TO ID ENDIF
      ."  " PC . ."   " NN MN @ WRCHR NN 1+ MN @  WRCHR
      NN 2+ MN @ WRCHR
         ID WRCHR ."   " X 1+ C@ . ." ," X 3 + C@ 8 SHL
         X 2+ C@ + . CR ;
: CKBP P 0< IF 1 TO STOP ELSE BP 0 > IF BP 0 DO
      P I BREAK @ = IF CR ." BREAK " P TO PC PCODE 1 TO STOP
      ENDIF LOOP ENDIF ENDIF ;
( DEFINITION OF STRING CONSTANTS FOR MAIN )
: 'R' 82 ;  : 'S' 83 ;  : 'X' 88 ;  : 'G' 71 ;
: 'T' 84 ;  : 'K' 75 ;  : 'B' 66 ;  : 'C' 67 ;
: 'Y' 89 ;  : 'E' 69 ;  : 'U' 85 ;  : 'N' 78 ;
: 'Q' 81 ;       -->
```

```
SCR # 69
( BEGIN \MAIN\ OF PASCAL INTERPRETER )
: MAIN CR ." P-CODE START ADDRESS IN HEX?  "  RDHEX TO Z
CR INIT P TO PC PCODE 0 TO BP
BEGIN ." >" RDCHR DUP WRCHR CR
CASE 'R' OF 0 TO STOP BEGIN EXEC CKBP STOP UNTIL ENDOF
      'S' OF EXEC P TO PC PCODE ENDOF
      'X' OF ." P=" P . ." B=" B . ." T=" T . ." S(T)=" S(T) .
             ." S(T-1)=" T 1- S @ . CR ENDOF
      'G' OF INIT BEGIN EXEC CKBP STOP UNTIL ENDOF
      'T' OF ." *TRACE*" CR U 0 DO
             TP 1+ DUP TO TP U 1- > IF 0 TO TP ENDIF
             TP TRACE @ 0 >= IF TP TRACE @ TO PC PCODE ENDIF
         LOOP ENDOF
     'K' OF ." ? " RDDEC DUP 7 + SWAP DO
             ."  " I S @ . CR LOOP ENDOF
     'B' OF BP BPLIM < IF BP 1+ DUP TO BP . ." :"  -->

SCR # 70
( \MAIN\ CONTINUED )
           RDDEC BP 1- BREAK ! CR ENDIF ENDOF
     'C' OF 0 TO BP CR ENDOF
     'Y' OF BP 0 > IF BP 0 DO ."  " I BREAK @ . CR LOOP ENDIF
           ENDOF
     'E' OF ." ? " RDDEC DUP TO P0 TO PC PCODE ENDOF
     'U' OF P0 0 > IF P0 1- DUP TO P0 TO PC PCODE ENDIF ENDOF
     'N' OF P0 1+ DUP TO P0 TO PC PCODE ENDOF
     'Q' OF -1 TO P  ENDOF
      ."  UNRECOGNIZED COMMAND " CR
ENDCASE ( OF CMD)
P 0 < UNTIL
CR K . ." INSTRUCTIONS EXECUTED " CR ;
```

**RDDEC** accepts a decimal number from the console as input to the stack, using **RDCHR**. Similarly, **RDHEX** accepts a hexadecimal number. As in any language, these routines are hardware dependent and must be modified by the reader to suit his system.

The procedure **SBASE** (screen 66) is used in lieu of the Pascal function BASE. The colon definitions **INIT** (screen 62), **EXEC** (screens 63 through 67), **PCODE** (screen 68) and **CKBP** (screen 68) are direct translations of their Pascal counterparts. **MAIN** (screen 69 and 70) is the super-procedure in Forth which emulates the MAIN body of the Pascal listing. Finally, the names of the variables have been kept pretty much the same to facilitate comparison of the listings. They are declared in screen 60.

### Using the Forth P-Code Interpreter

The Forth interpreter is self contained. Unlike the Pascal interpreter, it requires no explicit run-time support package, since it is completely embedded in, and supported by, Forth. Note, however, that one could reduce Forth to the minimum kernel required to run this interpreter. This residue would then be entirely analogous to the Pascal run-time support package.

In this writer's system, Forth occupies memory from 2D00H to 9000H and is supported by the North Star DOS located at 2000H up to 2A00H. Since the p-code to be interpreted is totally relocatable, it may be loaded anywhere below 2000H or above 9000H in the writer's system. Note that for a Pascal system which writes to memory, as does that in figure one, precautions must be taken to avoid writing over the p-code itself or into the region of DOS or Forth.

To invoke the interpreter once it has been typed into Forth, simply type **MAIN**. From there on, the interpreter behaves exactly like the Pascal version. Figure four is a partial example of its use on the p-code generated from the program in figure one. The reader should refer to the original *BYTE* magazine articles for further details on use of the interpreter.

```
      3 :BEGIN T:=T-1;S[T]:=S[T]-S[T+1] END;
      4 :BEGIN T:=T-1;S[T]:=S[T]*S[T+1] END;
      5 :BEGIN T:=T-1;S[T]:=S[T] DIV S[T+1] END;
     6 :S[T]:=S[T] AND 1; /TEST FOR ODD/
      7 :BEGIN T:=T-1;S[T]:=S[T] MOD S[T+1] END;
      8 :BEGIN T:=T-1;S[T]:=S[T]=S[T+1] END;
      9 :BEGIN T:=T-1;S[T]:=S[T]<>S[T+1] END;
      10:BEGIN T:=T-1;S[T]:=S[T]<S[T+1] END;
      11:BEGIN T:=T-1;S[T]:=S[T]>=S[T+1] END;
      12:BEGIN T:=T-1;S[T]:=S[T]>S[T+1] END;
      13:BEGIN T:=T-1;S[T]:=S[T]<=S[T+1] END;
      14:BEGIN T:=T-1;S[T]:=S[T] OR S[T+1] END;
      15:BEGIN T:=T-1;S[T]:=S[T] AND S[T+1] END;
      16:S[T]:=NOT S[T];
      17:BEGIN T:=T-1;S[T]:=S[T] SHL S[T+1] END;
      18:BEGIN T:=T-1;S[T]:=S[T] SHR S[T+1] END;
      19:S[T]:=S[T]+1;
      20:S[T]:=S[T]-1;
      21:BEGIN /COPY/
          T:=T+1;S[T]:=S[T-1] END
         ELSE BEGIN WRITE(' ILLEGAL OPR');CRLF;STOP:=1 END
      END /CASE OF A/;
   2:BEGIN /LOAD/
      L:=MEM[X+1];
      IF L=255 THEN S[T]:=MEM[S[T]]
      ELSE BEGIN IF IDX THEN A:=A+S[T];
            T:=T+1-IDX;S[T]:=S[BASE(L)+A] END
    END;
   3:BEGIN /STORE/
      L:=MEM[X+1];
      IF L=255 THEN BEGIN
       MEM[S[T-1]]:=S[T];T:=T-2 END
      ELSE BEGIN
       IF IDX THEN A:=S[T-1]+A;
       S[BASE(L)+A]:=S[T];T:=T-1-IDX END
    END;
   4:BEGIN /CALL/
      L:=MEM[X+1];
      IF L=255 THEN BEGIN CALL(S[T]);T:=T-1 END
      ELSE BEGIN
      S[T+1]:=BASE(L);S[T+2]:=B;
      S[T+3]:=P;B:=T+1;P:=A END
   END;
   5:IF T>(SIZE1-A) THEN BEGIN
     WRITE(' STACK OVFL');CRLF;STOP:=1 END
     ELSE T:=T+A;
   6:P:=A; /JMP/
   7:BEGIN IF S[T]=MEM[X+1] THEN P:=A; /JPC/
      T:=T-1 END;
   8:CASE A OF /CSP/
     0:BEGIN T:=T+1;READ(S[T]) END; /IN CHAR/
```

```
   1:BEGIN WRITE(S[T]);T:=T-1 END; /OUT CHAR/
   2:BEGIN T:=T+1;READ(S[T]#) END; /IN NUMBER/
   3:BEGIN WRITE(S[T]#);T:=T-1 END; /OUT NUMBER/
   4:BEGIN T:=T+1;READ(S[T]) END; /IN HEX/
   5:BEGIN WRITE(S[T]);T:=T-1 END; /OUT HEX/
   8:BEGIN /OUT STRING/
      FOR IDX:=T-S[T] TO T-1 DO WRITE(S[IDX]);
      T:=T-S[T]-1 END
   ELSE BEGIN WRITE(' ILLEGAL CSP');CRLF;STOP:=1 END
   END /CASE OF A/
ELSE BEGIN WRITE(' ILLEGAL OPCODE');CRLF;STOP:=1 END
 END /CASE OF F/
END /EXEC/;
PROC CODE(PC); /PRINT CODE/
 VAR X,N,IDX:INTEGER;
 BEGIN X:=PC SHL 2+Z;N:=MEM[X]*3;
  IF N<=24 THEN IDX:=' '
  ELSE BEGIN N:=N-48;IDX:='X' END;
  WRITE('  ',PC#,'  ',MN[N],MN[N+1],MN[N+2],IDX,'  ',
   MEM[X+1]#,',',MEM[X+3] SHL 8 +MEM[X+2]#);CRLF
 END /CODE/;
PROC CKBP; /CHECK BREAK POINT/
 VAR I:INTEGER;
 BEGIN IF P<0 THEN STOP:=1
  ELSE BEGIN
  FOR I:=1 TO BP DO
   IF BREAK[I]=P THEN BEGIN
    WRITE(' BREAK ');CODE(P);
     STOP:=1 END END
 END /CKBP/;
BEGIN /MAIN/
 FOR I:=0 TO 26 DO
  MN[I]:=MEM[I+1E80]; /MNEMONICS ARE IN MEMORY/
 WRITE('ADDR?');READ(Z);CRLF;
 INIT;CODE(P);BP:=0;
 REPEAT WRITE('>');READ(CMD);
  CASE CMD OF
'R':BEGIN STOP:=0;REPEAT EXEC;CKBP UNTIL STOP END;
'S':BEGIN EXEC;CODE(P) END;
'X':BEGIN
     WRITE('  P=',P#,' B=',B#,' T=',T#,
           ' S[T]=',S[T]#,' S[T-1]=',S[T-1]#);CRLF
    END;
'G':BEGIN INIT;REPEAT EXEC;CKBP UNTIL STOP END;
'T':BEGIN WRITE(' *TRACE*');CRLF;
     FOR I:=0 TO U DO BEGIN
      TP:=TP+1;IF TP>U THEN TP:=0;
      IF TRACE[TP]>=0 THEN CODE(TRACE[TP]) END
    END;
'K':BEGIN READ(I#);
     FOR J:=I TO I+6 DO
```

```
      WRITE('   ',S[J]#);CRLF
    END;
 'B':IF BP<BPLIM THEN BEGIN
       BP:=BP+1;WRITE(BP#,':');
       READ(BREAK[BP]#);CRLF END;
 'C':BEGIN /CLEAR BP/
     BP:=0;CRLF END;
 'Y':BEGIN FOR I:=1 TO BP DO
     WRITE('   ',BREAK[I]#);CRLF END;
 'E':BEGIN READ(P0#);CODE(P0) END;
 'U':IF P0>0 THEN BEGIN
       P0:=P0-1;CODE(P0) END;
 'N':BEGIN P0:=P0+1;CODE(P0) END;
 'Q':P:=-1
    ELSE BEGIN WRITE('SYNTAX ERROR');CRLF END
 END /CASE OF CMD/
 UNTIL P<0;
 CRLF;WRITE(K#,' INSTR. EXECUTED');CRLF
 END. /MAIN/
```

```
*LF MEM1.P,2 0800     (REMEMBER MEM1.P USES 0000 TO 001A)
*GO FORTH
****FORTH V1.02*****


MAIN
P-CODE START ADDRESS IN HEX?  0800


0  JMP 0,1          (REDUNDANT - CAN ELIMINATE)
>S                  (ONE STEP THE PROGRAM)
 1  INT 0,31
>S
 2  LIT 0,0
>B                (SET A BREAK POINT)
1 :5
>X             (DISPLAY STATUS)
P=2 B=1 T=31 S(T)=91 S(T-1)=90
>Y              (DISPLAY THE BREAKPOINT #'S)
5
>T                (DO A TRACE UP TO HERE)
**TRACE**
0  JMP 0,1
1  INT 0,31
>R                 (RUN TILL THE BREAKPOINT)
BREAK  5  OPR 0,21
>R              (GO AGAIN)
                (GOT FIRST LETTER O.K.)
BREAK  5  OPR 0,21
>C                (CLEAR THE BREAKPOINT)
>R                (GO FOR BROKE !)
BCDEFGHIJKLMNOPQRSTUVWXYZ[               (WHOOPEE ! ! )


555 INSTRUCTIONS EXECUTED
OK
```

**Figure Four**
**Example use of interpreter**

# Recursion

*Michael Ham*
*Santa Cruz, California*

A recursive definition uses in the definition itself the idea being defined. For example, consider the definition of the mathematical operator ! (pronounced "factorial"); the definition is usually stated by defining the general term n! ("n factorial"):

| | | |
|---|---|---|
| n! = | 1 | if n = 1 |
| = | n * (n-1)! | if n > 1 |

Although recursive definitions look circular, they are not, for the implied procedure does not lead to an infinite regression. Recursive definitions consist of two parts: in one part, the actual result is given for a specific value; in the other part—the recursive part—the idea being defined is used, but for a term smaller than the original term. This diminution of terms ultimately leads to the specific value defined in the first part.

In the example above, each application of the procedure gives a factorial number smaller (by one) than the number before; this ultimately leads to 1!, for which the definition provides the actual value. To see how the procedure works, use the definition to derive 4!:

| | |
|---|---|
| 4! = 4 * 3! | from the definition |
| = 4 * 3 * 2! | applying the definition to 3! |
| = 4 * 3 * 2 * 1! | applying the definition to 2! |
| = 4 * 3 * 2 * 1 | since 1!= 1 |
| = 24 | multiplying |

Recursive definitions are succinct and also imply an operational algorithm. Some computer languages (notably LISP) make extensive use of recursion. Forth can also use recursion, but first it must address the problem of a definition using itself.

For an example, consider the problem of finding the greatest common divisor (gcd) of two positive integers—that is, the greatest integer that divides evenly into both of them, with no remainder. The gcd of 8 and 9 is 1; the gcd of 8 and 12 is 4; and the gcd of 8 and 24 is 8.

Euclid long ago discovered that the gcd of two numbers—call them a and b—is also the gcd of b and a mod b. This reduces the problem of finding the gcd of two numbers to one of finding the gcd of two smaller numbers.

Moreover, the gcd of any positive integer and zero is the integer: for example, the largest integer that goes into 9 and 0 is 9, since *every* integer divides evenly into zero.

We thus can offer a recursive definition of the greatest common divisor of two nonnegative integers, a and b:

| | | |
|---|---|---|
| GCD (a,b) = | a | if b = 0 |
| = | GCD (b, a mod b) | if b > 0 |

This definition is easily translated into a Forth definition:

```
: GCD ( a b --- gcd ) ?DUP IF DUP ROT ROT
  UMOD GCD THEN ;
```

The **?DUP** checks to see whether b is already zero; if it is, then we are done: the greatest common divisor is a, and it is left alone on the stack when the **IF** eats the (unduplicated) zero (namely b) and control passes over the **IF THEN** clause.

If b is *not* zero, it is necessary only to execute **UMOD**, since we then shall have a mod b left on the stack. But we need to keep b around for the next step, and **UMOD** will use up the only copy, so it is first necessary to **DUP** b. After the **DUP** the stack is out of order, but **ROT ROT** straightens it up so that everything is ready for **UMOD**—and after **UMOD** executes, the stack contains only the two numbers b and a mod b, with the latter on top of the stack.

Note in passing that **SWAP OVER** more efficiently accomplishes the same things as **DUP ROT ROT**; for this reason **SWAP OVER** is used in the definition below. Some Forths achieve the same results with a single (though nonstandard) word **TUCK**.

The definition tells us that the gcd of b and a mod b will also be the gcd of the original pair (a and b). Since the two numbers on the top of the stack are exactly the two numbers we need and they are, moreover, in the right order, we need only to execute **GCD** again.

Here, unfortunately, a problem arises. If we try to enter the above definition, the Forth compiler will stop, confounded, with a message something like "? GCD." Since we are still in the middle of the definition, **GCD** is not found in the dictionary search.

The compiler very properly doesn't use the current definition: since Forth allows words to be redefined, the compiler should look for a previous definition for any word used in the current definition. Some mechanism must be used to ignore the current name in a dictionary search. In fig-FORTH, a bit (the "smudge" bit) is set in the header of the word currently being defined; this bit tells **-FIND** to ignore this word. The bit is toggled by ; when the definition is complete, and the word then can be found on subsequent dictionary searches.

But here we do want the current word's compilation address plugged into the definition sequence. Recall the structure of a Forth definition, using **GCD** as an example and letting "ac" stand for "compilation address" (see figure one).

Except for **IF**, which sets up a branch, the body of the definition consists of the compilation addresses of the words used. What we want to have in place at "x" is the compilation address of **GCD** itself, so that **GCD** will execute. Is there any way to get that address?

Many Forths include the word **LATEST**, which puts on the stack the address of the name field of the most recent definition. Some use the word **LAST**. **LAST** (or **LATEST**) can instead be a variable that contains the name field address of the most recent definition; in that case, you need the sequence **LAST @** to get the name field address itself on the stack.

The address of the name field must then be converted to the compilation address—for example, in fig-FORTH the sequence **PFA CFA** will do the job: **PFA** converts the address of the name field to the address of the parameter field, and

**CFA** converts the address of the parameter field to the address of the code field, which in fig-FORTH is the compilation address. (There is no word to go directly from the name field address to the code field address.) Once the address of the code field is on the stack, one can use , to store it into the dictionary. The definition we have arrived at is often named **MYSELF**, although the 83-Standard provides the name **RECURSE**. In fig-FORTH, this definition, thus, is:

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

The sole remaining consideration is that **MYSELF** must be made immediate, as shown above. That is, we don't want **MYSELF**'s compilation address to be stored in **GCD**; instead, we want **MYSELF** to *execute* during compilation of **GCD** so that **MYSELF** will pick up **GCD**'s name field address, convert it to **GCD**'s code field address, and put that address into the definition. And that is exactly what **IMMEDIATE** words do: they execute at once, even during compilation (when normal words are merely compiled for later execution).

With **MYSELF**, we can rewrite the definition of **GCD**:

```
: GCD ( a b --- gcd ) ?DUP IF SWAP OVER
  UMOD MYSELF THEN ;
```

Charles Moore has suggested a different approach, using a word he named **RECURSIVE**, which enables the current word to be found. In fig-FORTH, for example, **RECURSIVE** would simply clear the smudge bit. The definition can then use its own name to store its compilation address in the definition, as:

```
: GCD RECURSIVE ?DUP IF SWAP OVER
  UMOD GCD THEN ;
```

Note that ; must now clear (rather than toggle) the smudge bit when the definition is complete. If the bit was already cleared by **RECURSIVE**, clearing the cleared bit is simply a null operation, whereas a toggle would reset the bit and make the definition effectively vanish.

**Assignment for the Reader**

0. Figure out how to write a version of **MYSELF** in the Forth you use. (Check first to see if it already has **MYSELF** or **RECURSE**.) As indicated above, you want to obtain somehow the compilation address of the word being defined and during compilation store it into the dictionary at the appropriate spot. The compilation address is usually (though not always) the address of the code field.

1. Use your version of **MYSELF** to write a recursive Forth definition of **FACTORIAL** which will replace the top of the stack with its factorial value.

2. Use a **BEGIN UNTIL** structure to write a nonrecursive definition of **GCD**: that is, explicitly test the value of the remainder at each step. (Recursive definitions are often not the most efficient approach in terms of machine resources.)

3. Write a nonrecursive definition of **FACTORIAL**.

4. Is **RECURSIVE** immediate? Explain why or why not.



```
--------------------------------------------------------------------
! Head  !  link  !  GCD's ac  !  : run-time  ! ?DUP's ac !
--------------------------------------------------------------------

--------------------------------------------------------------------
  IF branch ! SWAP's ac  !  OVER's ac ! MOD's ac !  x  . . .
--------------------------------------------------------------------
```

**Figure One**

# SELECTED PUBLICATIONS

The FORTH Interest Group Order Form (*on the reverse side of this page*) has 5 newly added publications selected by the FIG Publications Committee:

Bibliography of Forth References, 2nd Edition
Journal of Forth Applications and Research, V. 2, #1
Mastering Forth
1984 Rochester Proceedings
**1983 FORML Proceedings**

*Here are brief descriptions of 2 of them:*

**A Bibliography of Forth References**
**Second Edition, September 1984**
*Thea Martin, editor*

The second edition of *A Bibliography* has over 1300 references to Forth related papers, books, and articles, from the US and abroad. Indexed by subject and author, *A Bibliography* also classifies references into relative levels — introductory, intermediate, or advanced. This year, complete publisher information has been added, and the subject index has been expanded

*A Bibliography of Forth References* was compiled as a service to the Forth community by The Institute for Applied Forth Research. Forth users around the world have contributed references to work in many countries and languages, from the early astronomy papers to the latest Japanese Forth Computer project.

William F. Ragsdale has called *A Bibliography of Forth References* "an invaluable aid", which "should be part of the library of any serious Forth user."

**MASTERING FORTH**
*by Anderson and Tracy*

A step-by-step tutorial to the high level, stack oriented Forth computer language. Formerly entitled FORTH TOOLS, this unique guide introduces you to each of the commands required by the Forth 83 International Standard — the preferred dialect of the Forth Interest Group. This book also includes utilities and extensions that can be written within the standard.

Because forth is an interactive language, this book is ideal for use while sitting at the computer. Inside you will find complete discussions on:

- stack manipulation
- variables
- loops
- strings
- compiling words
- defining words...and more.

# FORTH INTEREST GROUP
# MAIL ORDER FORM

NAME ____________________

COMPANY ____________________

STREET ____________________

CITY ____________________ STATE/PROV ____________ ZIP ____________

COUNTRY ____________________ TELEPHONE ( ) ____________________

| | PRICES US/FOREIGN AIR | |
|---|---|---|
| **Membership in the FORTH Interest Group & Volume 6 of FORTH Dimensions** | $15/27 | ______ |
| Volume 1 FORTH Dimensions | 15/18 | ______ |
| Volume 2 FORTH Dimensions | 15/18 | ______ |
| Volume 3 FORTH Dimensions | 15/18 | ______ |
| Volume 4 FORTH Dimensions | 15/18 | ______ |
| Volume 5 FORTH Dimensions | 15/18 | ______ |
| **BOOKS ABOUT FORTH** | | |
| All About FORTH | $25/35 | ______ |
| Beginning FORTH | 17/21 | ______ |
| FORTH Encyclopedia | 25/35 | ______ |
| FORTH Fundamentals, V. 1 | 16/20 | ______ |
| FORTH Fundamentals, V. 2 | 13/16 | ______ |
| Starting FORTH (Soft Cover) | 18/22 | ______ |
| Starting FORTH (Hard Cover) | 23/28 | ______ |
| Thinking FORTH (Soft Cover) | 16/20 | ______ |
| Thinking FORTH (Hard Cover) | 23/28 | ______ |
| Threaded Interpretive Languages | 23/28 | ______ |
| Understanding FORTH | 3/5 | ______ |
| **REFERENCE** | | |
| FORTH 83 Standard | $15/18 | ______ |
| FORTH 79 Standard | 15/18 | ______ |
| **CONFERENCE PROCEEDINGS** | | |
| FORML Proceedings 1980 | $25/35 | ______ |
| FORML Proceedings 1981 (2 V.) | 40/55 | ______ |
| FORML Proceedings 1982 | 25/35 | ______ |
| Rochester Proceedings 1981 | 25/35 | ______ |
| Rochester Proceedings 1982 | 25/35 | ______ |
| Rochester Proceedings 1983 | 25/35 | ______ |
| JOURNAL OF FORTH APPLICATIONS AND RESERACH | | |
| Journal of FORTH Research V. 1 #1 | $15/18 | ______ |
| Journal of FORTH Research V. 1 #2 | 15/18 | ______ |
| **REPRINTS** | | |
| Byte Reprints | $3.50/5 | ______ |

| | PRICES US/FOREIGN AIR | |
|---|---|---|
| Popular Computing 9/83 | $3.50/5 | ______ |
| Dr. Dobb's 9/81 | 3.50/5 | ______ |
| Dr. Dobb's 9/82 | 3.50/5 | ______ |
| Dr. Dobb's 9/83 | 3.50/5 | ______ |
| Dr. Dobb's 9/84 | 3.50/5 | ______ |
| **HISTORICAL DOCUMENTS** | | |
| Kitt Peak Primer | $25/35 | ______ |
| fig-FORTH Intallation Manual | 15/18 | ______ |
| ASSEMBLY LANGUAGE SOURCE LISTINGS | | |
| 1802 | $15/18 | ______ |
| 6502 | 15/18 | ______ |
| 6800 | 15/18 | ______ |
| 6809 | 15/18 | ______ |
| 68000 | 15/18 | ______ |
| 8080 | 15/18 | ______ |
| 8086/88 | 15/18 | ______ |
| 9900 | 15/18 | ______ |
| ALPHA MICRO | 15/18 | ______ |
| Apple II | 15/18 | ______ |
| ECLIPSE | 15/18 | ______ |
| IBM/PC | 15/18 | ______ |
| NOVA | 15/18 | ______ |
| PACE | 15/18 | ______ |
| PDP-11 | 15/18 | ______ |
| VAX | 15/18 | ______ |
| Z80 | 15/18 | ______ |
| T-Shirt Size: ______ | $10/12 | ______ |
| Poster (BYTE Cover) | 3/5 | ______ |
| Handy Reference Card | FREE | ______ |

**SUBTOTAL** ______

CA Residents Add 6½% Sales Tax ______

**TOTAL** ______

☐ VISA ☐ Mastercard # ____________________ Expiration Date ____________

$15 Minimum On VISA/Mastercard Orders.
All Prices Include Shipping.

Make Check or money order payable in US funds drawn on a US Bank to: FIG.
PAYMENT MUST ACCOMPANY ALL ORDERS (Including Purchase Orders).

**OFFICE USE ONLY**

By ____________ Date ____________ MO ______ TO ______ PU ______ Auth No ____________

Shipped By ____________ Date ____________ Weight ____________ UPS ______ USPS ______

Hold ____________ Date ____________ Weight ____________ UPS ______ USPS ______

ORDER PHONE: (408) 277-0668

FORTH INTEREST GROUP • P.O. BOX 8231 • SAN JOSE, CA 95155

# Forth Semaphores

*Jens Zander*
*Linkoping, Sweden*

Using parallel or concurrent processes or tasks is very often a natural solution to a programmer's problem. Although convenient, these solutions may result in quite complex systems. Even if each task is of low complexity, the total number of possible states our systems can be in grows very fast with the number of tasks (in fact, exponentially). Very quickly we lose control of the system as a whole. In these cases it is of vital importance that we can isolate the tasks from each other to avoid unwanted interference.

Multi-tasking systems are essentially of two kinds, task-controlled systems or "true" concurrent (time-shared) systems. In the task-controlled systems, task switching is entirely up to the tasks themselves. Each task has to decide when it would be better to let some other task take control of the system. In a "true" concurrent system, however, task switching is performed by the system itself. From the user's point of view, this system is easy to use. Programs to be executed by these machines are written in the same way as for single-user systems. The only difference is that each task will execute slower. The concurrent solution is, of course, the only possible one in a multi-user system. Forth lends itself very nicely to the implementation of both these schemes.

Isolating the different tasks makes them easy to handle since they do not interfere. Nevertheless, after having gone through all the trouble of making the tasks act as independent entities, we will be forced to consider the problem of making them cooperate. Consider the following example:

```
: ADD _COUNTER
  COUNTER @
  1+ COUNTER !
;
```

Several processes may simultaneously use this word to modify the common variable **COUNTER**. In a task-driven system this will cause no problems. Each task will increment **COUNTER** by one each time **ADD_COUNTER** is executed and will not follow any other task to interfere before finishing. In a "true" concurrent system, however, the different processes will not be aware of each other. Of course, most of these systems are not *truly* concurrent. What is important, however, is that every activity may be interrupted in favour of another at virtually any instant. In our example two processes may simultaneously read the contents of **COUNTER**. After adding one and simultaneously writing the result, we end up with an increment of one instead of two. Other examples where problems of this kind will arise are the sharing of I/O devices and data transfer between processes. The bottom line in all of these situations is that the processes need to become aware of each other in these situations. We need some kind of mutual exclusion and synchronization mechanism.

## Semaphores

Various methods of solving problems of this kind have been invented. In the late sixties, Dijkstra proposed an elegant solution by introducing the concept of *semaphores* [5,6,2]. We will introduce a Forth version of the semaphores and the two primitives **WAIT** and **SIGNAL** used to manipulate them. **WAIT** and **SIGNAL** are modifications of Dijkstra's functions P and V. (Please see figure one.)

| | | |
|---|---|---|
| **SEMPHORE** | **CCCC** ( --- ) | |
| **CCCC** : | ( --- addr) | |

Creates the semaphore **CCCC** with initial content of one. **CCCC** will return the address of the semaphore.

| | | |
|---|---|---|
| **WAIT** | ( --- addr) | **C** |

Checks the contents of the semaphore at addr. If the contents of the semaphore is non-zero it is decremented by one. If the semaphore is zero, execution is suspended until the semaphore contents become non-zero. Non-sharable word.

| | |
|---|---|
| **SIGNAL** | ( addr --- ) |

Increments the content of the semaphore at addr by one. Non-sharable word.

**Figure One**

A semaphore is an ordinary memory cell or variable, accessible to all involved processes. **SEMAPHORE** is used to create a variable of this kind. The semaphore will be initialized to contain a one. **WAIT** will take a semaphore address as an argument and will check the contents of the semaphore cell. If the content is non-zero, **WAIT** will decrement it by one. If the content is found to be zero, execution will be suspended until the contents becomes non-zero again. **SIGNAL** will just increment the semaphore, regardless of its value. The critical feature of these words is that they are *exclusive* or non-sharable. This means that they in some way cannot be interrupted by the system scheduler, and thus cannot be simultaneously executed by two processes. Due to this, we may avoid the problem encountered in the **ADD_COUNTER** example above. With the primitives **WAIT** and **SIGNAL** we make whole sequences of words exclusive. Figure two is a modified version of **ADD_COUNTER** including semaphores. This version will now work properly without unwanted interference between tasks. We note that the initial value of the semaphore (one is the default value) will determine the number of tasks that will be allowed to enter the exclusive section. Any additional tasks will be suspended by **WAIT** until some of the involved processes exit the section and execute the **SIGNAL** operation. Processes **WAIT**ing will form a *semaphore queue*. This way they will not steal time from active processes.

Besides the excluding function, semaphores may be used for handshaking during data transfer between tasks. Handshaking is basically a matter of synchronization. Figure three is an example of this kind. Here, characters are passed between two processes using the words **SENDER** and **RECEIVER**. The initial value of the semaphore **RX.READY** will tell the transmitting process how many characters he will accept before he has to start processing them with **RECEIVE_CHARACTER**. On the other hand, the semaphore **TX.READY** will reflect the number of characters transmitted but not yet received. In situations like this we have to watch out carefully for the ever present menace of deadlock. Deadlocks occur mainly when communicating tasks get out of phase. The reader may try to figure out what will happen if the communication link is initialized with both **RX.READY** and **TX.READY** equal to zero.

### Implementing Semaphores

Since **WAIT** and **SIGNAL** are to be exclusive words, their implementation will heavily depend on how multi-tasking is achieved. In general we have to lower to the level where the task switching is performed, in order to be able to implement an excluding function. In hardware multi-tasking functions with multiple physical processors, this will call for some hardware solution. The most common multi-tasking implementations, however, are those implemented in some host machine using the native code of the machine. The multi-tasking Forth kernel will provide the user with several virtual Forth machines. A machine code kernel and scheduler handles the task switching and will thus also be able to handle the semaphores. The simplest way to achieve an exclusive function is probably to disable the timer interrupt controlling the task switching. This may, however, not be 100% effective in all cases. In the following we will give an example of sema-

```
SEMAPHORE  TX.READY
SEMAPHORE  RX.READY
0 TX.READY !
:    SENDER
       BEGIN
          RX.READY WAIT
          SEND_CHARACTER
          TX.READY SIGNAL
       AGAIN
;

:    RECEIVER
       BEGIN
          TX.READY WAIT
          RECEIVE_CHARACTER
          RX.READY SIGNAL
       AGAIN
;
```

**Figure Three**

```
SEMAPHORE  COUNTER.SEM

: ADD_COUNTER
      COUNTER.SEM WAIT
      COUNTER   1+ COUNTER ! (Protected section)
      COUNTER.SEM SIGNAL
;
```

**Figure Two**

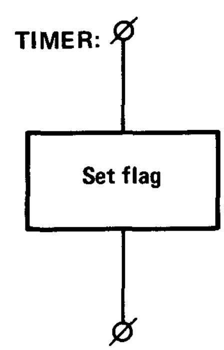



**Figure Four-a**

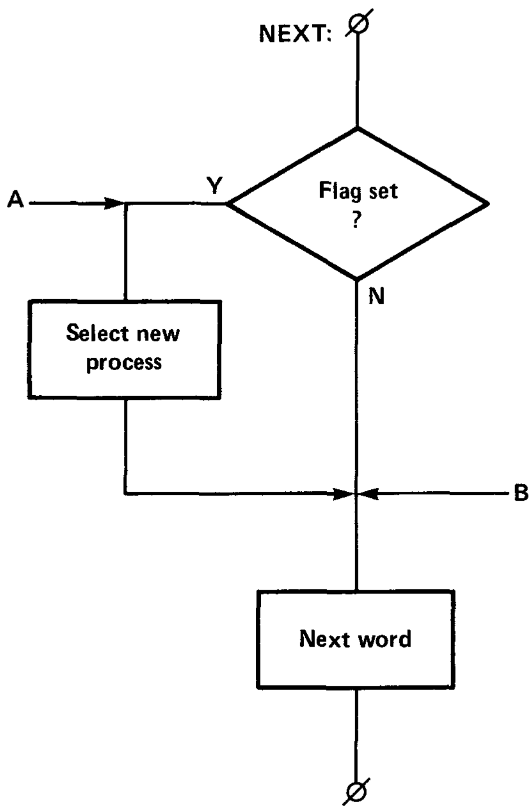



Figure Four-b

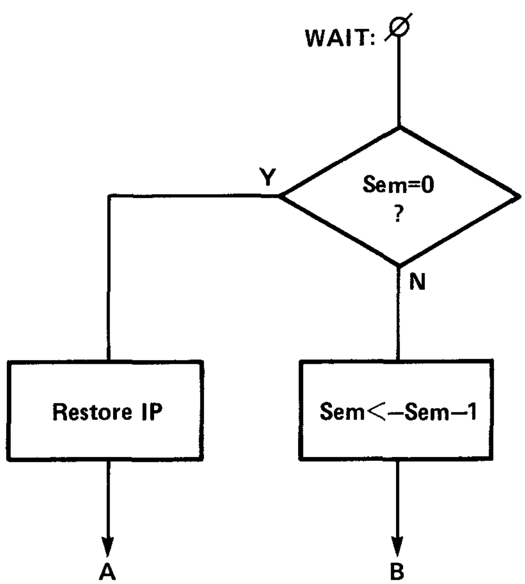



Figure Four-c

phore and kernel implementions as found in MFORTH used at the University of Linkoping[3]. The kernel flowcharts are found in figure four. A host machine timer interrupt is used to initiate process switching. The interrupt does not force a process switch, it will only set a flag. This flag will tell the Forth virtual machine that it may change processes whenever ready to do so. The flag is checked by the inner interpreter (**NEXT**). If the flag is set, the scheduler is activated to select a new task. This task is loaded, and execution will continue with this task during the next time slice.* Task switching occurs only between high-level Forth words. This means that code words (the "machine" instructions of the Forth-machine **) are never interrupted. If we want to implement an exclusive instruction (e.g. **WAIT**) we simply use a code definition. In order to save time, **WAIT** will force a task switch each time a zero semaphore is encountered. This is a simple way to implement a semaphore queue. The processes **WAIT**ing are not really suspended, they will check the semaphore each time they are activated by the scheduler, and then "go to sleep" by issuing a **SWITCH** if the semaphore is still zero.

### High-Level Semaphores

There may be situations when we would like to implement semaphores without lowering to the task switch level. An example of this is a hardware multiprocessor system with no support hardware for semaphores. A typical case is processors transparently sharing memory. To solve this problem we may use a neat trick from carrier sense random

* At this point a Forth process switch is done very fast. Only the internal registers of the Forth machine (**IP**, **RP**, **SP** and **UP**) have to be saved and restored. The scheduling overhead is, therefore, quite low in a system like this. When loaded with five tasks of different priorities, a typical MFORTH system (1 MHz 6809, 20 ms time slices) will spend only 1.5% of the total time scheduling. The scheduler consists of less than 100 bytes of code.

** In fact, using the flag is nothing other than implementing a Forth machine interrupt. As in ordinary microprocessors, the current machine instruction is finished before the interrupt is acknowledged.

access communications as found in LANs (Ethernet) and packet radio systems.

To implement a mutual exclusion device, we will, instead of using one semaphore cell, use a boolean vector with one cell for each of the processes involved. When a process has been granted exclusive execution, its vector cell will contain a true value. A process needing exclusive execution will first sense, or scan the vector, for any true values. If a true value is found in the vector, the process has to wait until the vector is all false. If the vector is found all false the processor will raise its own flag, i.e. make it true. This is, however, not sufficient to exclude all other processes. Figure five explains why. During the time interval between sensing the vector to be empty or false and setting the flag, some other task may have started its exclusion sequence. To make the exclusion safe, we have to check the vector again after some time interval $t_d$. Figure five shows the worst case. We can see that $t_d$ has to be at least as large as the longest sense-to-set interval in all processes. After this delay, we sense again. If a collision occurs, i.e. two tasks are simultaneously requesting exclusive rights, we choose one of them by some arbitrary non-ambiguous rule. In the Forth implementation shown in figure six we will choose the one with the lowest index **TASK-ID**. The word **SENSE** will leave either the index of the lowest true flag in the **STATE-VECTOR** or a zero if no true flag is found. **PROTECT** and **UNPROTECT** form the framework of an exclusive program section. Note the use of **SWITCH** which is used to force a process switch (cf. **PAUSE** in ref. 1) to save time. **PROTECT** and **UNPROTECT** are used to implement **WAIT** and **SIGNAL**. We may, however, use them directly to produce some exclusive section. One should note that, in this case, no other exclusive program section may be executed in the system.

The high-level semaphores offer a very useful system-dependent task synchronization mechanism. Their major drawback is an elaborate procedure with quite slow execution, especially if many tasks are involved.

**References**


1. Laxen, H., "Multi-Tasking, Part 1," *Forth Dimensions* V/4.

2. Brinch-Hansen, P., *Operating System Principles*, Prentice-Hall, 1973.

3. Zander, J., "Multi-Tasking FORTH Implementation for the 6809, Users Manual," Internal Report LiTH-ISY-I-0577, Dec. 1982.

4. Tsichritzis, D.C., Bernstein, P.A., *Operating Systems,* Academic Press, New York, 1974.

5. Dijkstra, E.W., "Cooperating Sequential Processes" in *Programming Languages* (F. Genuys ed.) Academic Press, New York, 1968.

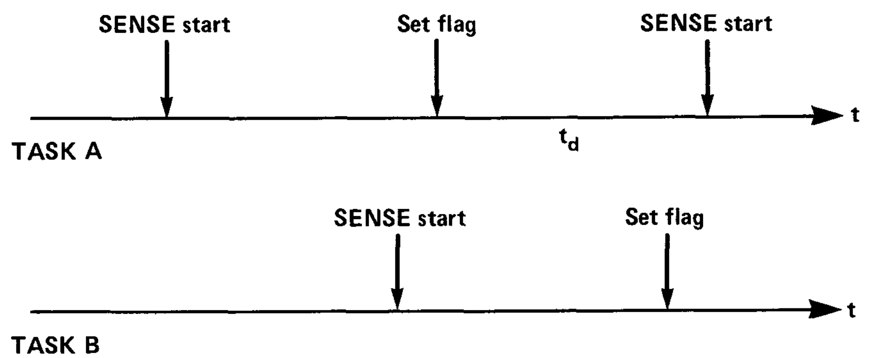




**Figure Five**

```
( High Level Semaphores  JZ 22Feb84 Forth-79 )

10 CONSTANT #TASKS      56 USER TASK-ID

CREATE (SV)  #TASKS ALLOT

: STATE-VECTOR 1- (SV) + ;

: CLEAR STATES  #TASKS 1+ 1 DO
                0 I STATE-VECTOR C! LOOP ;
:  SENSE ( --- n )
         0  #TASKS 1+ 1 DO
                          I STATE-VECTOR C@
                          IF DROP I LEAVE THEN
                        LOOP ;
-->
```

```
( High Level Semaphores  JZ 22Feb84 Fo th-79 )

: SET-STATE 1 TASK-ID @ STATE-VECTOR C  ;
: CLEAR-STATE 0 TASK-ID @ STATE-VECTOR C! ;

( System dependent Delay word )
:  DELAY 40 0 DO LOOP ;

: PROTECT  BEGIN SENSE NOT
              IF SET-STATE DELAY  SENSE
                 TASK-ID @ = IF EXIT THEN
              THEN
              CLEAR-STATE  SWITCH
            AGAIN ;
: UNPROTECT CLEAR-STATE ;
-->
```

```
( High Level Semaphores  JZ 22Feb84 Forth-79 )

: SIGNAL PROTECT 1 SWAP +! UNPROTECT ;

: WAIT BEGIN  PROTECT   DUP @ 0=
       WHILE  UNPROTECT SWITCH
       REPEAT
       -1 SWAP +! UNPROTECT ;

EXIT
```

**Figure Six**

# Forth-83 Program to Run Forth-79 Code

*Robert Berkey*
*Palo Alto, California*

As the Forth-83 Standard becomes more widely adopted, there is an increasing need to translate Forth-79 programs into Forth-83. This article contains a translator program that allows a Forth-79 program to run on a Forth-83 system; additionally, words that are difficult to translate automatically are discussed. The article focuses on the required word set of the Forth-79 Standard and program requirements of the Forth-83 Standard.

In most respects a Forth-83 system is a superset of a Forth-79 system. It is therefore possible to run a Forth-79 program on a Forth-83 system by placing a translator between the program and the system. For most Forth-79 words the translation is trivial.

Such a translator offers several benefits, the main one being the ability to run Forth-79 programs in a Forth-83 environment. Although this is practical, there is a speed penalty for compute-bound programs. Most primitive words implemented in the translator run at about one-third the speed of their code equivalents. The majority of Forth-79 words are unchanged in Forth-83 and run at full speed, so the overall speed penalty is roughly 50%. For many applications this is acceptable. In addition to the speed penalty, a program with the translator will, of course, require more memory than the program alone.

The translator is also useful as a programmer's tool for changing a Forth-79 program into a Forth-83 program. The entire translator, including the loops and math, can be loaded onto a Forth-83 system and the program to be converted loaded on top of that. Piece-by-piece the program can be upgraded and the corresponding part of the translator removed.

### Simple Translations

The first group of words, shown in figure one, are the simplest in translation.

### Division

If the application's divisions result in negative quotients and the remainders are not zero, the floored division in Forth-83 must be converted to the rounded-to-zero division used in Forth-79. See figure two.

### Forth-79 Editing Words

In the unlikely event that the 79-Standard program uses **SCR** and/or **LIST** and these are not available in the system, the code in figure three will suffice.

### Do-Loops

Certain uses relating to do-loops do not translate directly between Forth-79 and Forth-83. These include:

- Unusual do-loop parameters, especially for the n1 n1 **DO ... LOOP** case when used in Forth-79 to execute the loop once.
- **LEAVE**

```
FORTH-83 DECIMAL   FORTH DEFINITIONS
: FORTH-83   ( — )   1 ABORT" Not Forth-83" ;   ( replaceable, see below )
: FORTH   ( — )   FORTH ; IMMEDIATE            (            "   , see below )
: 79-STANDARD   ( — )   ." Not fully Forth-79" ;     (  "   , see below )
: 0<   ( n — flag )   0< NEGATE ;
: 0=   ( w — flag )   0= NEGATE ;
: 0>   ( n — flag )   0> NEGATE ;
: <   ( n1 n2 — flag )   < NEGATE ;
: =   ( w1 w2 — flag )   = NEGATE ;
: >   ( n1 n2 — flag )   > NEGATE ;
: D<   ( d1 d2 — flag )   D< NEGATE ;
: U<   ( u1 u2 — flag )   U< NEGATE ;
: NOT   ( 16b1 — 16b2 )   0= ;
: PICK   ( n — 16b )   1- PICK ;
: ROLL   ( n — 16b )   1- ROLL ;
: ."   ( <ccc> -- )   STATE @ IF   [COMPILE] ."
     ELSE   34 WORD COUNT TYPE   THEN ; IMMEDIATE
: ?   ( addr — )   @ . ;
: MOVE   ( addr1 addr2 n -- )   0 MAX   DUP + >R   OVER OVER - 1+
   R> SWAP IF   CMOVE
   ELSE   ?DUP IF   0 DO   OVER I + @   OVER I + !   2 +LOOP
     THEN   DROP DROP
   THEN ;
: CMOVE   ( addr1 addr2 n -- )   0 MAX   CMOVE ;
   ( Note: Redefine after defining  MOVE )
: CONSTANT   ( — 16b )   ( creating: 16b — )   CREATE ,
   DOES> @ ;   ( This allows a  CONSTANT  to be ticked )
: EXPECT   ( addr n -- )   0 MAX   OVER SWAP EXPECT
   SPAN @ +   0 SWAP C! ;
: FILL   ( addr n 8b — )   SWAP 0 MAX SWAP   FILL ;
: FIND   ( — addr )   32 WORD   FIND   0= IF   DROP 0   THEN ;
: '   ( — addr )   ' >BODY
   STATE @ IF   [COMPILE] LITERAL   THEN ; IMMEDIATE
: KEY   ( -- c )   KEY 127 AND ;
: LITERAL   ( — 16b )  ( compiling: 16b — )
   STATE @ IF   [COMPILE] LITERAL   THEN ; IMMEDIATE
: QUERY   ( — )   TIB 80 EXPECT   SPAN @ #TIB ! ;
   ( Note: Define after redefining  EXPECT )
: SPACES   ( n -- )   0 MAX SPACES ;
: TYPE   ( addr n — )   0 MAX TYPE ;
: U*   ( u1 u2 — ud1 )   UM* ;
: U/MOD   ( ud u1 — u2 u3 )   UM/MOD ;
```

**Figure One**
**Simple Translations**

Problems with do-loop parameters are not frequent and the changes involved with **LEAVE** are more cleanly and simply done by rewriting the code into Forth-83. Still, a reasonably efficient Forth-79 loop and leave can be written in Forth-83. **LOOP** executes six extra primitives per iteration, so an empty loop will run seven times slower. An application loop with six primitive words in the loop body will run at half speed. See figure four.

### Forth-83 Standard Programs

Vocabulary considerations prevent this translator from being labeled a Forth-83 Standard Program. A Forth-83 Standard Program cannot redefine standard words in the vocabulary **FORTH**, but the basic form of this translator extensively redefines the standard words. The translator is otherwise standard, meeting both the portability and documentation requirements of the standard. See Appendix A for the Forth-83 program documentation. The vocabulary option in Appendix B avoids the redefinition problem at the expense of portability.

### Automatic Translation Limitations

The translator has given the code that translates directly from Forth-83 to Forth-79, but not all of the Forth-79 system requirements can be satisfied. For the remaining translation problems there may be no easy fixes. Words and usages difficult to translate automatically are:

**CURRENT** This will be available if the Forth-83 system has the System Extension Word Set.

**CONTEXT** Even if the Forth-83 system has the System Extension Word Set, this might require some tinkering. This is a result of increased variety of vocabulary mechanisms in today's systems.

**FORGET** This now uses the compilation vocabulary, not the dictionary search order.

**EMPTY-BUFFERS** The phrase **SAVE-BUFFERS EMPTY-BUFFERS** can be replaced with **FLUSH**, but **EMPTY-BUFFERS** is no longer supported by the standard. Like **SCR** and **LIST**, **EMPTY-BUFFERS** tends to be around in a system and also tends not to be used by a program—it is really for programmers hacking at the keyboard.

**COMPILE** An uncommon usage, of the form **COMPILE [ 0 , ]** will not translate directly; alternate programming techniques or system-dependent surrogates may be needed for this case.

**WORD** The delimiter stored at the end of the text is now always a space. A 79-Standard program that uses a delimiter other than a space will require special handling to get running.

**Multi-programming impact** This was not fully specified in Forth-79. If a program does something such as typing out of a block buffer, it will have to be modified to be portable; on a single-task system this nonstandard practice should continue to work without problems.

For additional information on modifying a Forth-79 program to run on a Forth-83 system see the preceding issue of *Forth Dimensions*, "Upgrading Forth-79 Programs To Forth-83".

```
: ROUND-TO-ZERO    ( mod quo1 — quo2 )
   SWAP IF   DUP 0<   IF   1+   THEN THEN ;
: MODQUO>REMQUO    ( mod quo1 divisor — rem quo2 )   >R   DUP 0<
   IF   OVER IF   1+ SWAP R@ - SWAP   THEN   THEN   R> DROP ;
( The above words are not in Forth-79 but are used to develop the )
( standard words. )
: */    ( n1 n2 n3 — n4 )   */MOD   ROUND-TO-ZERO ;
   ( Caution: Redefine  */  before redefining  */MOD )
: */MOD    ( n1 n2 n3 — n4 n5 )   DUP >R   */MOD
   R> MODQUO>REMQUO ;
: /    ( n1 n2 — n3 )   /MOD   ROUND-TO-ZERO ;
   ( Caution: Redefine  /  before redefining  /MOD )
: MOD    ( n1 n2 — n3 )   DUP >R   /MOD   0< IF
   DUP IF   R@ -   THEN THEN   R> DROP ;
   ( Caution: Redefine  MOD  before redefining  /MOD )
: /MOD    ( n1 n2 — n3 n4 )   DUP >R   /MOD
   R> MODQUO>REMQUO ;
```

**Figure Two**
**Division**

```
CREATE TYPE-BUFFER   64 ALLOT
( The above word is not in Forth-79 but is used to develop the )
( standard words. )
VARIABLE SCR
: LIST    ( screen# — )    16 0 DO    CR    DUP BLOCK I 64 * +
    TYPE-BUFFER 64 CMOVE    TYPE-BUFFER 64 TYPE    SPACE I .    LOOP
    SCR ! ;
```

**Figure Three**
**Forth-79 Editing Words**

### Appendix A:
### Forth-83 Program Documentation Requirements

In most respects this translator program qualifies for being labeled a Forth-83 Standard program. However, it is non-standard because standard words are redefined in the **FORTH** vocabulary and the redefinitions do not comply with the Forth-83 Standard. It does meet the portability requirements of the standard and should work on any Forth-83 Standard System. Additionally, with the following documentation, the program satisfies the documentation requirements for a Forth-83 Standard Program. For additional information on documentation requirements see the Forth-83 Standard, p. 13.

- Dictionary space used: minimum required, 88 bytes; typical indirect threaded system, 1440 bytes.
- Largest use of data stack for any one word: minimum required, 10 bytes.

```
( First a leave-flag stack is created. )
CREATE LEAVE-FLAGS   HERE 22 + ,   20 ALLOT
   ( Allows room for ten nested loops. )
   ( The item second from top on the leave stack will be true )
   ( if a negative going  +LOOP  should terminate.  The item  )
   ( on the top of the leave stack is true if a positive going )
   ( +LOOP  or a  LOOP  should terminate. )
0 CONSTANT FALSE   -1 CONSTANT TRUE
: DON'T-LEAVE   ( — )   -1 LEAVE-FLAGS +!
   FALSE LEAVE-FLAGS @ C! ;
: DO-LEAVE   ( — )   -1 LEAVE-FLAGS +!
   TRUE LEAVE-FLAGS @ C! ;
: +LEAVE?   ( — ? )   LEAVE-FLAGS @ C@ ;
: -LEAVE?   ( — ? )   LEAVE-FLAGS @ 1+ C@ ;
: +-LEAVE?   ( n — n ? )
   DUP 0< IF   -LEAVE?   ELSE   +LEAVE?   THEN ;
: BEGIN-LOOP   ( n1 n2 — n1 n2 )   ( Set up initial leave )
   ( flags for this loop )   OVER OVER >
   IF   DO-LEAVE DON'T-LEAVE   ELSE   DON'T-LEAVE DO-LEAVE
   THEN ;
: END-LOOP   ( — )   2 LEAVE-FLAGS +! ;
   ( The above words are not in Forth-79 but are used to develop )
   ( the standard words. )
: DO        COMPILE    BEGIN-LOOP
            [COMPILE] DO ; IMMEDIATE
: LOOP      COMPILE    +LEAVE?
            [COMPILE] IF
            [COMPILE] LEAVE   ( a Forth-83  LEAVE )
            [COMPILE] THEN
            [COMPILE] LOOP
            COMPILE    END-LOOP ; IMMEDIATE
: +LOOP     COMPILE    +-LEAVE?
            [COMPILE] IF
            COMPILE    DROP
            [COMPILE] LEAVE   ( a Forth-83  LEAVE )
            [COMPILE] THEN
            [COMPILE] +LOOP
            COMPILE    END-LOOP ; IMMEDIATE
: LEAVE   ( — )   2 LEAVE-FLAGS +!   DO-LEAVE DO-LEAVE ;
   ( Caution: Redefine  LEAVE  after redefining  LOOP  and  +LOOP )
```

**Figure Four**
**Do-Loops**

```
VOCABULARY FORTH-79
: 79-STANDARD   ( — )   FORTH-83   FORTH-79 DEFINITIONS
   ." Not fully Forth-79" ;
79-STANDARD DECIMAL
: FORTH-83   ( — )   FORTH FORTH-83 ;
   ( preserve access to the 83 definitions )
: FORTH   ( — )   FORTH-79 ; IMMEDIATE
```

**Figure Five**

- Largest use of return stack for any one word: minimum required, 2 bytes; typical indirect threaded system, 8 bytes.
- Mass storage blocks: seven screens of Forth source—no specific block ranges required.
- Operator's terminal facilities: no special requirements.

### Appendix B
### Preserving the Forth-83 System Words

It may be preferred to keep the Forth-83 definitions available. This can conveniently be done if the Forth-83 system being used can have more than two vocabularies in the search order. (Most can and in fact do.) The code in figure five would replace the definitions in figure one for **FORTH-83**, **FORTH**, and **79-STANDARD**, and puts subsequent definitions in a vocabulary called **FORTH-79**. These definitions will not work as intended on some standard systems and therefore do not meet the portability requirements of the standard.

# ANDIF and ANDWHILE

*Wendall C. Gates, PE*
*Santa Cruz, California*

Anyone who works in real-time, control-oriented programming frequently encounters the need to implement decisions based on several input conditions. Forth implements single-condition branching as **IF ELSE THEN** and **BEGIN WHILE REPEAT** statements, but multiple-condition branching is absent in most Forth implementations.

One extremely simple approach, which solves both the multiple **IF** and multiple **WHILE** applications, is presented on screens 78 and 79 (fig-FORTH). **ANDIF** is used in the form:

**...IF...ANDIF...ANDIF...ANDIF...ELSE...THEN**

where **ELSE** is optional and the number of **ANDIF**s is not constrained. The compile-time action of **ANDIF** is to compile first **DUP** and **0BRANCH**. Then the second entry on the computation stack (the address of the word following the **0BRANCH** compiled by **IF**) is copied over the first entry (the compiler security digit) and is decremented by two, becoming the address of the first **0BRANCH**. This address is then compiled. The final action is to compile **DROP**. In other words, each **ANDIF** compiles a **0BRANCH** which points back to the **0BRANCH** compiled by **IF**; thus, only one forward branch needs to be compiled, and it is handled by **ELSE** or **THEN** as usual.

At run time, the flag being tested is duplicated. If the duplicate copy of the flag is true, the original flag is dropped and execution continues inline. If the flag is false (zero), **ANDIF**'s **0BRANCH** branches back to **IF**'s **0BRANCH**; the original flag then directs **IF**'s **0BRANCH** to skip forward to **ELSE** or **THEN**.

```
SCR # 78
  \ ANDIF  Multiple IF  Statement                       WCG  7-7-84
  : ANDIF
    ?COMP  DUP  2 ?PAIRS                          \ Compiler security
    COMPILE DUP  COMPILE 0BRANCH     \ duplicate flag, 0branch back
                                  \ to IF, then out to ELSE or THEN
    OVER   2- ,          \ address of first 0BRANCH is second stack
                           \ entry, under compiler security, minus 2
    COMPILE DROP ;                    \ if flag true, drop duplicate
    IMMEDIATE     -->

     ;-----------------------------------------------;
   0BRANCH (addr) <next test, leave flag> DUP 0BRANCH (addr) DROP

   This code directs the false exit(s) back through the first
   0BRANCH (compiled by IF); therefore, the ELSE...THEN part of
   the conditional branching still work as usual.


SCR # 79
  \ ANDWHILE   Multiple WHILE Statement                 WCG  7-7-84
  : ANDWHILE
    ?COMP  DUP  4 ?PAIRS                          \ Compiler security
    COMPILE DUP  COMPILE 0BRANCH     \ duplicate flag, 0branch back
                                  \ to WHILE, then out past REPEAT
    OVER   2- ,          \ address of first 0BRANCH is second stack
                           \ entry, under compiler security, minus 2
    COMPILE DROP ;                    \ if flag true, drop duplicate
    IMMEDIATE     ;S

     ;-----------------------------------------------;
   0BRANCH (addr) <next test, leave flag> DUP 0BRANCH (addr) DROP

   This code directs the false exit(s) back through the first
   0BRANCH (compiled by WHILE); therefore, the REPEAT part of the
   conditional branch structure still works as usual.
```

**ANDWHILE** is similarly constructed (screen 79); in fact, the only difference is the compiler security digit. Usage is identical to **ANDIF**; no matching closeout words (**ENDWHILE** in Ref. 1) are needed to resolve the branching.

**ANDIF** and **ANDWHILE** also permit building complex control structures in a simple, straightforward fashion. Here, for example, is a multi-condition, multi-step structure using **ANDIF**:

test1
**IF** task1 test2
**ANDIF** task2 test3
**ANDIF...**
**ELSE...**
**THEN**

In this sequence, each test leaves a flag. The sequence of tasks will be executed until a test leaves a false flag, at which point execution will jump to the code following **ELSE** (if used) or following **THEN**. Note that the code following **ELSE** will not be executed at all if all conditions test true, but will be executed if any condition tests false. Tasks must leave the stacks unaltered.

A multi-conditional, multi-step loop can be programmed as:

**BEGIN**
test1
**WHILE** task1 test2
**ANDWHILE** task2 test3
**ANDWHILE...**
**REPEAT**

This code will loop through the sequence of tasks until a test leaves a false flag; execution then jumps immediately out of the loop to the code following **REPEAT**.

This technique of directing all unsuccessful exits out through the original **0BRANCH** imposes both a speed penalty and a code-size penalty over methods which compute and store back the exit address. The extra words are all primitives, so the speed penalty is small. The size penalty is four bytes per branch, balanced by savings in the code needed to implement **ANDIF** and **ANDWHILE** versus a heavier-duty solution (for example, the **IT ENDIT** code presented by Luoto in Ref. 3).

**References**


1. Hayden, Julian. "Multiple WHILE Solution," *Forth Dimensions* III/3, p. 72.

2. Harris, Kim. "Transportable Control Structures," 1981 Forth Standards Conference, pp. 97-107.

3. Luoto, Kurt. "Parnas' it...ti Structure," *Forth Dimensions* VI/1, pp. 26-31.

*FORTH Dimensions*

# Index to Volume Five

*This reference guide to Volume V was prepared as a service for our readers and for all members of the Forth Interest Group. Items are referenced by issue and page number. The first entry, for example, refers to an article on 3-D Animation which appeared in Volume V, Issue 1, page 11.*

# Mixing CODE With High-Level Forth

*Henry Laxen*
*Berkeley, California*

One of Forth's nicest features is the ability to easily integrate high-level, machine-independent Forth code with low-level, machine-dependent assembly code. This fact has many implications, the most significant of which is that the issue of run-time efficiency can usually be deferred until much of the application has been completed. Once a system reaches a certain critical mass, it is no longer intuitively obvious which routines to recode in order to improve performance; and much programmer time is generally wasted by trying to optimize procedures early in the game, before meaningful performance measurements can be gathered. Thus, performance improvement should be deferred as long as possible, until after the system is running and is no longer subject to massive change. The nice thing about Forth is that there is usually little or no penalty for this waiting period and, in fact, frequently no fine tuning is necessary. However, if that were always the case, this would be a very short and dull essay. The purpose of this paper is to examine some programming techniques that will simplify performance enhancement.

The problem that I propose to address is how to easily and conveniently mix high-level code with low-level code. In one direction, this problem is more or less solved by the use of **CODE** words. **CODE** is a Forth defining word which allows the user to use the assembly language of his machine to define a new word in Forth. Thus, on an 8080 the lines presented in figure one are functionally identical, even though the code version is about ten times faster.

Frequently, when it comes to speed optimization, there are a few critical functions which can be rewritten as **CODE** words and integrated into the system. This is usually all that is required. A different problem, which in reality doesn't occur very often, involves writing inline

```
: 2*   2 *   ;

CODE 2*    AX POP  AX AX ADD  AX PUSH  NEXT  END-CODE
```

Figure One

```
Header(DOUBLE)
runtime(NEST)
cfa(2)                      Header(2)
                            runtime(CONSTANT)
                            value(2)

cfa(*)                      Header(*)
                            Here+2 for code word
                            Assembly language code

cfa(.)                      Header(.)
                            runtime(NEST)
                            cfas of words called by .

cfa(UNNEST)                 Header(UNNEST)
                            Here+2 for code word
                            Assembly language code
```

Figure Two

```
: [C   ( -- )
   HERE 2+ ,   HERE 2+ ,   ASSEMBLER [COMPILE] [  ;
   IMMEDIATE

: C]   ( -- )
   [ ASSEMBLER ] HERE 6 + * IP MOV  NEXT  FORTH  ]   ;

: DOUBLE  ( n -- )
   [C AX POP  AX AX ADD  AX PUSH  C]  .  ;
```

Figure Three

```
LABEL HILEVEL
  RP DEC  RP DEC  IP 0 [RP] MOV  IP POP  NEXT
: C:   ( -- )
   HILEVEL #) CALL  FORTH  ]   ;
CODE (;C)    ( -- )
   IP PUSH  0 [RP] IP MOV  RP INC  RP INC  RET  END-CODE
: ;C  ( -- )
   [ ASSEMBLER ] COMPILE (;C)  ASSEMBLER
   [COMPILE] [   ;    IMMEDIATE

CODE EXAMPLE   ( -- )
   5 # AX MOV  AX PUSH  C:  ." High Level" DUP  . ;C
   AX POP  NEXT  END-CODE
```

**Figure Four**

assembly language code within a high-level definition. While you could always make a separate **CODE** word out of the inline assembly language and then simply reference the word, I think it would be an interesting intellectual exercise to see just how we could accomplish this inline if we wish.

The idea is that while we are executing high-level code, we all of a sudden want to run some inline assembly language code and then return to high level when we are done. A high-level word has a parameter field that contains pointers to code fields. The code fields themselves contain pointers to code. See figure two for an illustration of this.

We know that when we are running in a high-level definition, the IP (interpretive pointer) is inside the parameter field of the current word being executed. It is pointing at a word which contains the code field address of the next component word to be executed. This code field points to machine-executable code. In general, the CFA of **CODE** words points two bytes beyond itself, which is where the actual code begins. Thus, to create inline code, we must duplicate this structure. This is accomplished with the code presented in figure three.

Notice that we must do the **HERE 2+** , twice. The first one is a pointer to a code field; the second is the code field that points to code, which is inline. Next we go into the **ASSEMBLER** vocabulary, and stop compilation with [. This word must be immediate so that it is executed while we are compiling. The only mysterious thing about the definition of **C]** is the magic number six. Well, on an 8080, six is the number of bytes occupied by the **# MOV** and **NEXT #)** instructions, so it is what we must set the IP to in order to continue interpreting high-level code after the low-level inline code. We then reenter compilation by calling ]. Finally, **DOUBLE** is an example of how we would use **[C** and **C]**. We go immediately from high level to low level, double the number on the stack, and return to high level, where we print it out with . (dot).

I don't think writing inline assembly language code in your high-level definitions is very useful, and I don't recommend it; but I included it for your amusement and edification. The other direction, however, can be extremely useful, namely calling high-level definitions from assembly language and returning. One application of this comes to mind immediately: fetching or storing characters in an I/O buffer. If the buffer holds, say, 1K of characters, then only after calling it a thousand times do you actually need to perform I/O. The actual I/O operation is thus rather rare and usually involves executing a lot of code. Wouldn't it be nice if you could write a code word that usually fetches a character out of the buffer but, when the buffer is empty, calls a high-level word such as **BLOCK** to perform the necessary I/O. You could do this by factoring the character-by-character I/O into two pieces and passing flags back and forth, but this is inefficient and ugly. The inline high-level code solution is much cleaner. The code that implements this is shown in figure four.

Let's see if we can figure out how this works. The word **C:** assembles a **CALL** instruction and starts up the Forth compiler with ]. The **CALL** instruction pushes the address of the next word onto the stack, and then executes the code at the label **HILEVEL**. This code saves the current value of the IP on the return stack and sets the IP to the value that is currently on the parameter stack, which was left there by the **CALL** instruction. That is really all there is to it. We are now executing high-level Forth code inline. When we are through executing Forth code and want to return to the assembly language word we were called by, we end the high-level code with **;C**. This compiles **(;C)** inline and leaves the compiler to return to the assembler. At run time, **(;C)** pushes the current value of the IP and restores the old value of the IP from the return stack. Since **NEXT** has already incremented the IP to point to the next word, a simple **RET** instruction brings us back to the assembly language code that follows the **;C**. An example of how this is used is shown in the word **EXAMPLE**. We first load a 5 into the AX register and push it onto the stack. We then enter high-level code and print out the string "High Level" followed by the number on the parameter stack. We then return to the code word we were in, pop the stack and jump to **NEXT**. While the example does not perform a terrible useful function, it does illustrate the transition between low- and high-level code.

Although the code presented here is only for an 8080, if you understand the principles involved it will not be difficult for you to translate it for your processor if you want to use it. Once this kind of tool is available to you, I am sure you will find many applications for it. Implementing fast character-by-character I/O buffering is just one common application. It is also handy for displaying error messages during low-level hardware diagnostic code. Use your imagination! Anyway, that is all for now and, until next time, may the Forth be with you.

# Chapter News

*John D. Hall*
*Oakland, California*

**I still haven't heard!**

Would you like to talk to other people who use Forth? Would you like to hear how other people use Forth? Would you like to interest other people in the projects you are working on? Would you like to know more about Forth? If any of the answers were yes and you live near one of the missing cities (see sidebar), then why don't you get together with the others in your area and get a FIG chapter started? Each of these cities has more than enough FIG members to have a chapter, but where is it?

There are even more people than these FIG members. For each FIG member in your city (listed to the right of the city in the accompanying figure) there seem to be five other people who use Forth but who have not yet joined the Forth Interest Group. Two of those five probably are experts in Forth! Get them to join! Get them to help you!

I can't start the chapter for you. You will have to make the effort. How to do it? Try this for starters:

1) Write or call and tell me you are interested in trying.(I will send you a chapter kit and the names of the FIG members in your area.) Write to:

Forth Interest Group
Att'n: John D. Hall
P.O. Box 8231
San Jose, California 95155
or call the FIG Hotline: 415-962-8653

2) Decide on a temporary meeting time and place. Choose it at your convenience, since you are the one making the effort to get the chapter started.

3) Contact the other FIG members in your area by telephone, letter or a notice in your local computer newspaper. (Spend a little money on this one—you will get it back later.)

4) Call the first meeting! Discuss interests, then decide on a second meeting and format.

5) At the second meeting, a) elect officers and a program chairman (distribute the responsibilities, it makes life easier that way), b) collect dues for the chapter (repay yourself for the original out-of-pocket expenses), and c) estalish a list of speakers for the following meetings.

6) Have five FIG members sign the Chapter Certification Form and return it to me.

We have four new chapters, and one special interest group has changed to a chapter with meetings. That makes a total of sixty-five chapters!

Atlanta FIG Chapter
Atlanta, Georgia

New Orleans FIG Chapter
New Orleans, Louisiana

Detroit FIG Chapter
Detroit, Michigan

Austin FIG Chapter
Austin, Texas

East Tennessee FIG Chapter
Oak Ridge, Tennessee

**Atlanta FIG Chapter**

July 10: Our meeting was well attended; thanks to Computone for allowing us to use their excellent facilities. Alan Sandercock described an elegant Forth conversion of a *BYTE* magazine article about benchmarking of array multiplication. We are always interested in making comparisons with other languages. Alan, thanks for sharing your expertise with us. Talking about comparisons, as many of you know, Ada is now the standard for Department of Defense mission-critical software. Looking at many of the "unique" features of

Ada, such as "packages" of code that can be individually compiled and reused, one is reminded of how much of Forth we accept as normal that many in the software world are just beginning to appreciate. The future and strength of Ada is in the automated environment for lifecycle support. I sense that this is the key for future quality, productivity and cost reduction, and I worry that the Forth community may be missing the boat. As usual, some other topics surfaced and created a lively debate. This time we were concerned about the meaning of "real time" and the significance of interrupt handling. In my mind, real time means the ability to complete a process on behalf of an external system in such a way as to influence the external system. Real time normally, but not necessarily, means fast! An interrupt is more simply defined as a means to suspend a process in response to an external event in such a way that the process can be resumed.

—*Ron Skelton*

**Detroit FIG Chapter**

July 26: The July meeting was held at the Ford Diversified Products Technical Center. The first part of what is to be an ongoing discussion of the basics of Forth was started. The first topic was a short discussion of what the Forth languages is and isn't. Some of the most simple Forth words—e.g., **+ . DUP DROP SWAP * SPACE SPACES CR KEY EMIT @ ! ; VARIABLE CONSTANT .**" "—were discussed and demontrated. The "visible stack" feature of the Bay Area Atari Forth (public domain) was helpful while demonstrating data stack operation, although bugs in this version prevented us from using it throughout the meeting and APX Forth was later used. Basic concepts of the dictionary and the data stack were discussed. Colon definitions and sample Forth coding sheets were distributed. The "Large Letter F" program from *Starting Forth* was discussed and demonstrated. The tutorial was planned to continue through the August meeting with chapter two of *Starting Forth.*

—*Thomas Chrapkiewicz*

**East Tennessee FIG Chapter**

June 12: The East Tennessee Forth Interest Group (ET-FIG) was formed at its first meeting, with over twenty people in attendance. The meeting, which was held in Oak Ridge, featured three presentations by local FIG members. Dr. Ray Adams gave a very enjoyable and informative paper titled, "Why Forth, and What Forth is Good For Amidst Computer Languages." This was followed by brief presentations by Norman Smith and Richard Secrist reviewing available Forth literature and "Implementing fig-FORTH on the VAX-11 in PDP-11 Compatibility Mode."

—*Richard Secrist*

**Kansas City FIG Chapter**

June 26: Fourteen people attended the meeting. We discussed the pros and cons of the Forth-83 Standard. The 83-Standard is definitely an improvement, but some questioned the wisdom of changing a standard, especially at this time. Some also felt the standard does not encompass enough.

July 24: Twelve people attended. Terry Rayburn shared his experience of metacompiling into ROM. Bill Jellison is in the process of procuring equipment for the network. It will probably be at least three months before he is ready. Whether or not you feel there is a place for floating-point math in Forth, and whether or not you even have support for floating point, you will do a lot of calculations in Forth using fixed point. Terry Rayburn has recommended *Computer Approximations* by Hart to help you write those complicated algorithms in fixed point.

—*Linus Orth*

**Missing Cities!**

Huntsville, Alabama (5)
Anchorage, Alaska (5)
Fairbanks, Alaska (5)
(Kodiak has a chapter!)
Escondido, California (26)
Santa Barbara, California (25)
(Eight other California chapters!)
Gainesville, Florida (9)
Orlando, Florida (11)
Tampa, Florida (25)
Honolulu, Hawaii (9)
Chicago, Illinois (30)
Evansville, Indiana (5)
Lafayette, Indiana (6)
Ames, Iowa (5)
Rochester, Minnesota (6)
Lincoln, Nebraska (5)
Reno, Nevada (9)
Newark, New Jersey (42)
Las Cruces, New Mexico (5)
Santa Fe, New Mexico (7)
Buffalo, New York (8)
Charlotte, North Carolina (8)
Raleigh, North Caroline (9)
Nashua, New Hampshire (19)
Columbus, Ohio (7)
Toledo, Ohio (7)
Oklahoma City, Oklahoma (5)
Corvallis, Oregon (12)
Pittsburg, Pennsylvania (11)
Memphis, Tennessee (7)
El Paso, Texas (5)
San Antonio, Texas (7)
Salt Lake City, Utah (5)
Seattle, Washington (54)
Madison, Wisconsin (17)
Milwaukee, Wisconsin (17)

Copenhagen, Denmark (7)
Helsinki, Finland (8)
Tokyo, Japan (21)
Amsterdam, Netherlands (8)
Wellington, New Zealand (5)
Oslo, Norway (8)
Barcelona, Spain (5)
Stockholm, Sweden (5)

# FIG Chapters

## U.S.

### • ALASKA

**Kodiak Area Chapter**
Call Norman C. McIntosh
907/486-4843

### • ARIZONA

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

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

### • CALIFORNIA

**Berkeley Chapter**
Monthly, 2nd Sat., 1 p.m
10 Evans Hall
University of California
Berkeley
Call Mike Perry
415/644-3421

**Los Angeles Chapter**
Monthly, 4th Sat., 11 a.m.
Allstate Savings
8800 So. Sepulveda Boulevard
½ mile North of LAX
Los Angeles
Call Phillip Wasson
213/649-1428

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

**Orange County Chapter**
Monthly, 4th Wed., 7 p.m.
Fullerton Savings
Talbert & Brookhurst
Fountain Valley
Monthly, 1st Wed., 7 p.m.
Mercury Savings
Beach Blvd., & Eddington
Huntington Beach
Call Noshir Jesung
714/842-3032

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

**Sacramento Chapter**
Monthly, 2nd Tues., 7 p.m.
170B 59th St., Room C
Call Tom Ghormley
916/444-7775

**Silicon Valley Chapter**
Monthly, 4th Sat., 1 p.m.
Dysan Auditorium
5201 Patrick Henry Dr.
Santa Clara
Call Glenn Tenney
415/574-3420

**Stockton Chapter**
Call Doug Dillon
209/931-2448

### • COLORADO

**Denver Chapter**
Monthly, 1st Mon., 7 p.m
Call Steven Sarns
303/477-5955

### • CONNECTICUT

**Central Connecticut Chapter**
Monthly, 1st Thurs., 7 p.m.
Meriden Public Library
Call Charles Krajewski
203/344-9996

### • FLORIDA

**Southeast Florida Chapter**
Miami
Call John Forsberg
305/252-0108

### • GEORGIA

**Atlanta Chapter**
Call Ron Skelton
404/393-8764

### • ILLINOIS

**Central Illinois Chapter**
Urbana
Call Sidney Bowhill
217/333-4150

**Fox Valley Chapter**
Call Samuel J. Cook
312/879-3242

**Rockwell Chicago Chapter**
Call Gerard Kusiolek
312/885-8092

### • INDIANA

**Central Indiana Chapter**
Monthly, 3rd Sat., 10 a.m.
Call Richard Turpin
317/923-1321

**Fort Wayne Chapter**
Call Blair MacDermid
219/749-2042

### • IOWA

**Iowa City Chapter**
Monthly, 4th Tues.
Engineering Bldg., Rm. 2128
University of Iowa
Call Robert Benedict
319/337-7853

### • KANSAS

**Wichita Chapter (FIGPAC)**
Monthly, 3rd Wed., 7 p.m.
Wilbur E. Walker Co.
532 S. Market
Wichita, KS
Call Arne Flones
316/267-8852

### • LOUISIANA

**New Orleans Chapter**
Call Darryl C. Olivier
504/899-8933

### • MASSACHUSETTS

**Boston Chapter**
Monthly, 1st Wed.
Mitre Corp. Cafeteria
Bedford, MA
Call Bob Demrow
617/688-5661 after 7 p.m

### • MICHIGAN

**Detroit Chapter**
Call Tom Chrapkiewicz
313/562-8506

### • MINNESOTA

**MNFIG Chapter**
Even month, 1st Mon., 7:30 p.m.
Odd Month, 1st Sat., 9:30 a.m.
Vincent Hall Univ. of MN
Minneapolis, MN
Call Fred Olson
612/588-9532

### • MISSOURI

**Kansas City Chapter**
Monthly, 4th Tues., 7 p.m.
Midwest Research Inst.
Mag Conference Center
Call Linus Orth
816/444-6655

**St. Louis Chapter**
Monthly, 3rd Tues., 7 p.m.
Thornhill Branch of
St. Louis County Library
Call David Doudna
314/867-4482

### • NEVADA

**Southern Nevada Chapter**
Suite 900
101 Convention Center Drive
Las Vegas, NV
Call Gerald Hasty
702/452-3368

### • NEW MEXICO

**Albuquerque Chapter**
Call Rick Granfield
505/296-8651

### • NEW YORK

**FIG, New York**
Monthly, 2nd Wed., 8 p.m.
Queens College
Call Tom Jung
212/432-1414 ext. 157 days
212/261-3213 eves.

**Rochester Chapter**
Bi-monthly, 4th Sat., 2 p.m.
Hutchison Hall
Univ. of Rochester
Call Thea Martin
716/235-0168

**Syracuse Chapter**
Monthly, 1st Tues., 7:30 p.m.
Call C. Richard Corner
315/456-7436

### • OHIO

**Athens Chapter**
Call Isreal Urieli
614/594-3731

**Cleveland Chapter**
Call Gary Bergstrom
216/247-2492

**Cincinatti Chapter**
Call Douglas Bennett
513/831-0142

**Dayton Chapter**
Twice monthly, 2nd Tues., &
4th Wed., 6:30 p.m.
CFC 11 W. Monument Ave.
Suite 612
Dayton, OH
Call Gary M. Granger
513/849-1483

### • OREGON

**Greater Oregon Chapter**
Monthly, 2nd Sat., 1 p.m.
Computer & Things
3460 SW 185th, Aloha
Call Timothy Huang
503/289-9135

- **PENNSYLVANIA**

**Philadelphia Chapter**
Monthly, 3rd Sat.
LaSalle College, Science Bldg.
Call Lee Hustead
215/539-7989

- **TENNESSEE**

**East Tennessee Chapter**
Monthly, 2nd Tue., 7:30 p.m.
Sci. Appl. Int'l Corp, 8th Fl.
800 Oak Ridge Turnpike, Oak Ridge
Call Richard Secrist
615/482-9031

- **TEXAS**

**Austin Chapter**
Contact: Matt Lawernce
P.O. Box 180409
Austin, TX 78718

**Dallas/Ft. Worth Metroplex Chapter**
Monthly, 4th Thurs., 7 p.m.
Software Automation, Inc.
14333 Porton, Dallas
Bill Drissel
214/264-9680

**Houston Chapter**
Call Dr. Joseph Baldwin
713/749-2120

- **VERMONT**

**Vermont Chapter**
Monthly, 3rd Mon., 7:30 p.m.
Vergennes Union High School
Rm. 210, Monkton Rd.
Vergennes, VT
Call Hal Clark
802/877-2911 days
802/452-4442 eves

- **VIRGINIA**

**First Forth of Hampton Roads**
Call William Edmonds
804/898-4099

**Potomac Chapter**
Monthly, 1st Tues., 7 p.m.
Lee Center
Lee Highway at Lexington St.
Arlington, VA
Call Joel Shprentz
703/437-9218 eves.

**Richmond Forth Group**
Monthly, 2nd Wed., 7 p.m.
Basement, Puryear Hall
Univ. of Richmond
Call Donald A. Full
804/739-3623

## FOREIGN

- **AUSTRALIA**

**Melbourne Chapter**
Monthly, 1st Fri., 8 p.m.
Contact: Lance Collins
65 Martin Road
Glen Iris, Victoria 3146
03/29-2600

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

- **BELGIUM**

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

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

- **CANADA**

**Nova Scotia Chapter**
Contact: Howard Harawitz
227 Ridge Valley Rd.
Halifax, Nova Scotia B3P 2E5
902/477-3665

**Southern Ontario Chapter**
Monthly, 1st Sat., 2 p.m.
General Sciences Bldg.
Rm. 312
McMaster University
Contact: Dr. N. Solntseff
Unit for Computer Science
McMaster University
Hamilton, Ontario L8S 4K1
416/525-9140 ext. 2065

**Toronto FIG Chapter**
Contact: John Clark Smith
P.O. Box 230, Station H
Toronto, ON M4C 5J2

- **COLOMBIA**

**Colombia Chapter**
Contact: Luis Javier Parra B.
Aptdo. Aereo 100394
Bogota
214-0345

- **ENGLAND**

**Forth Interest Group — U.K.**
Monthly, 1st Thurs., 7 p.m., Rm. 408
Polytechnic of South Bank
Borough Rd., London
Contact: Keith Goldie-Morrison
Bradden Old Rectory
Towchester, Northamptonshire
NN12 8ED

- **FRANCE**

**French Language Chapter**
Contact: Jean-Daniel Dodin
77 rue du Cagire
31100 Toulouse
(16-61) 44.03

- **GERMANY**

**Hamburg FIG Chapter**
Monthly, 4th Sat., 1500 hrs.
Contact: Horst-Gunter Lynsche
Holstenstr. 191
D-2000 Hamburg 50

- **IRELAND**

**Irish Chapter**
Contact: Hugh Doggs
Newton School
Waterford
051/75757 or 051/74124

- **ITALY**

**FIG Italia**
Contact: Marco Tausel
Via Gerolamo Forni 48
20161 Milano
02/645-8688

- **REPUBLIC OF CHINA**

**R.O.C.**
Contact: Ching-Tang-Tzeng
P.O. Box 28
Lung-Tan, Taiwan 325

- **SWITZERLAND**

**Swiss Chapter**
Contact: Max Hugelshofer
ERNI & Co. Elektro-Industrie
Stationsstrasse
8306 Bruttisellen
01/833-3333

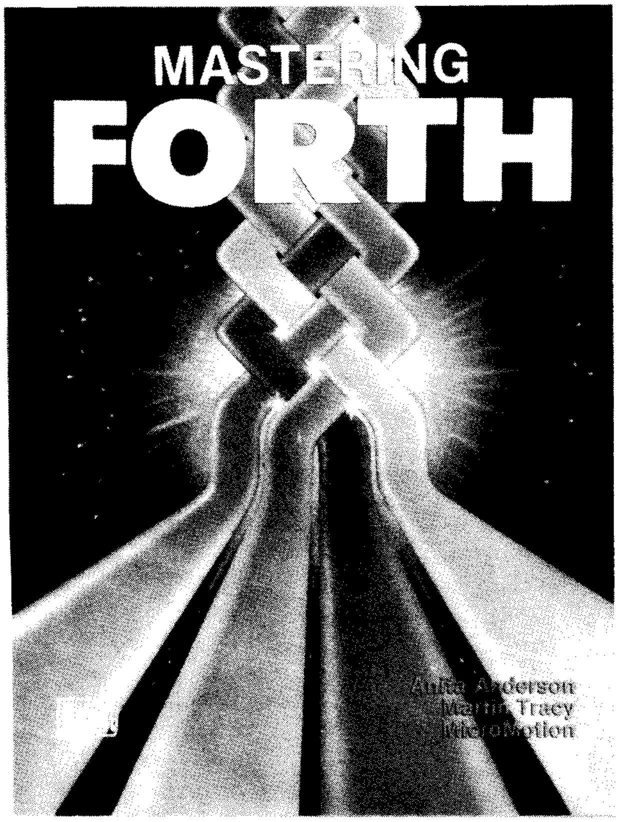
## SPECIAL GROUPS

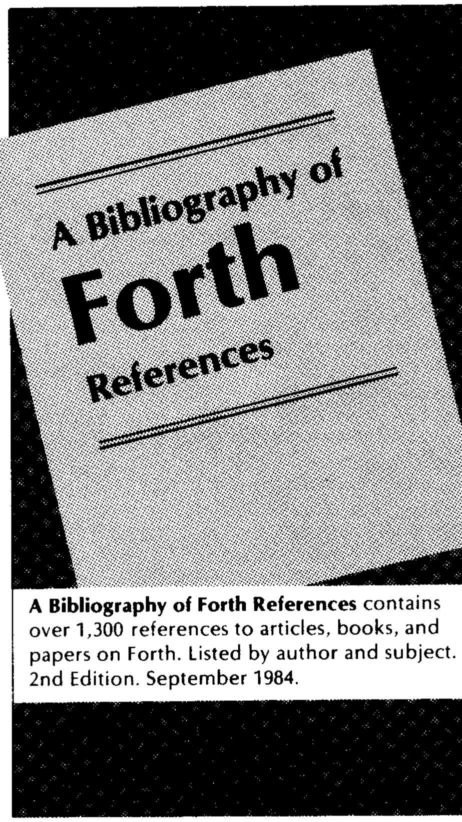
**Apple Corps Forth Users Chapter**
Twice Monthly, 1st & 3rd Tues., 7:30 p.m.
1515 Sloat Boulevard, #2
San Francisco, CA
Call Robert Dudley Ackerman
415/626-6295

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

**FIGGRAPH**
Call Howard Pearlmutter
408/425-8700

# ANNOUNCING






## ORDER FROM THE FORTH INTEREST GROUP

COMPLETE ORDER FORM ON PAGE 22

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

BULK RATE
U.S. POSTAGE
PAID
Permit No. 3107
San Jose, CA

ROBERT SMITH
2300 ST. FRANCIS DR.
PALO ALTO, CA 94303

Address Correction Requested