FORTH DIMENSIONS

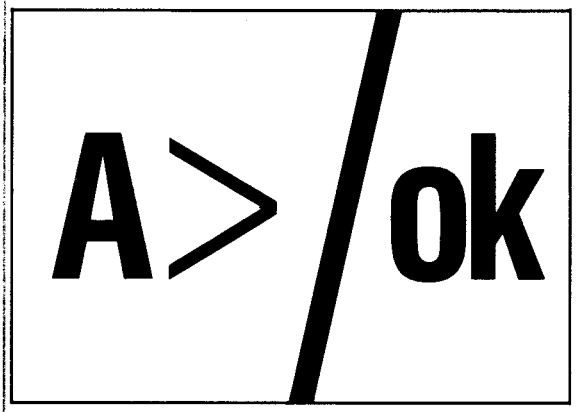
VOLUME IV, NUMBER 3 $2.50

INSIDE:

OPERATING SYSTEMS

FEATURES:

DEPARTMENTS:



OPERATING SYSTEMS

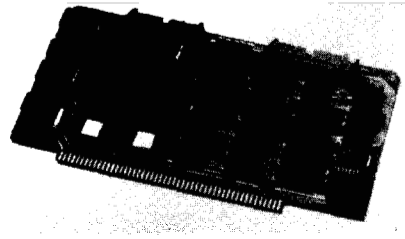
DEVELOPMENT TOOLS

Develop FORTH code for any target 8080/Z80 system on your current 8080/Z80 or Cromemco CDOS based system

8080/Z80 METAFORTH CROSS-COMPILER

- Produces code that may be downloaded to any Z80 or 8080 processor
- Includes 8080 and Z80 assemblers
- Can produce code without headers and link words for up to 30% space savings
- Can produce ROMable code
- 79 Standard FORTH
- Price $450

No downloading – No trial PROM burning. This port-addressed RAM on your S-100 host is the ROM of your target system



WORD/BYTE WIDE ROM SIMULATOR

- Simulates 16K bytes of memory (8K bytes for 2708 and 2758)
- Simulates 2708, 2758, 2516, 2716, 2532, 2732, 2564 and 2764 PROMS
- The simulated memory may be either byte or 16-bit word organized
- No S-100 memory is needed to hold ROM data
- Driver program verifies simulated PROM contents
- Price $495 each

CONSULTING SERVICES

Inner Access provides you with Custom Software Design. We have supplied many clients with both Systems and Application Software tailored to their specific needs. Contact us for your special programming requirements.

FORTH WORKSHOPS

ONE-WEEK WORKSHOPS — ENROLLMENT LIMITED TO 8 STUDENTS

FORTH Fundamentals

- Program Design
- Program Documentation
- FORTH Architecture
- FORTH Arithmetic
- Control Structures
- Input/Output
- The Vocabulary Mechanism
- Meta-Defining Words

OCT. 4-8 NOV. 8-12
JAN. 3-7 FEB. 7-11

$395 Incl. Text

Advanced FORTH Applications

- FORTH Tools
- Engineering Applications
- Floating Point
- Communications
- Sorting & Searching
- Project Accounting System
- Process Control
- Simulations

NOV. 15-19
FEB. 14-18

$495 Incl. Text

Advanced FORTH Systems

- FORTH Internals
- Assemblers and Editors
- Other Compilers
- Cross-Compilation Theory
- Romability, Multitasking, Timesharing
- File Systems/ Database Systems

OCT. 11-15
JAN. 10-14

$495 Incl. Text

Instructors: LEO BRODIE, GARY FEIERBACH and PAUL THOMAS

(For further information, please send for our complete FORTH Workshop Catalog.)

Inner Access Corporation

P.O. BOX 888 • BELMONT, CALIFORNIA 94002 • (415) 591-8295

Letters . . .

Cordic Erratum

Dear FIG,

Someone just pointed out an error in my algorithm outline, shown in Vol. IV, No. 1, page 14 (box), line 7: **"begin ANGLE <-ANGLE-2^{n-2}; XLAST<- -Y; YLAST<- -X end; ."**

The minus sign in front of the "X" should be deleted. The minus sign in front of the "X" on line 9 is correct.

Alan Furman
Palo Alto, CA

Pushing Frontiers

Dear FIG,

The Math Edition was one of the best issues I have ever read. While the commercial magazines have all turned very one dimensional (business), yours is still looking at new ideas and pushing the software frontier.

Kenneth B. Butterfield
Graduate Student
University of New Mexico

Thanks for the feedback, Ken, we'll try to stay Four Dimensional. —Editor

FORTH in College

Dear FIG,

This Fall, I will be teaching an introductory FORTH course at San Diego State. I will be running a modification of James' PDP-11 fig-FORTH on our VAX in RSX-11 compatibility mode. (The modifications will be mainly to make the FORTH follow *Starting FORTH*.) It's a bit strange to teach it on a time-sharing system, but we are embarrassingly short of micros. There will be some advantages, though: I will be able to distribute materials and consult with students over the phone net more easily than if we were using a roomful of micros.

I hope to have an article about my experiences teaching FORTH in time for your Teaching FORTH issue.

Vernor Vinge
Dept. of Math Sciences
San Diego State University

Good luck with your class. We'll be looking forward to your article. —Editor

Holland Hobbyists

Dear FIG,

I am in charge of software distribution in our 150+ member Hobby Computer Club here at Philips Data Systems, in Apeldoorn, Holland, and we have decided to add FORTH to our standard club-supported languages.

Our current "P1" (!) is based on 2650 processor, and we are now modifying an excellent 2650 FORTH implementation made by Dr.E. J. Hannivoort, here in Holland. We are now busy selecting a new CPU design for use this year (probably Z-80), and will of course be offering FORTH on the new design too.

Interest in FORTH here in Holland is well started — there was a local FORTH Interesse Groep (as we call it here in Holland!) meeting yesterday under Theo van Lottum's chairmanship, and they are providing an excellent range of information on FORTH. However, with more and more of our members asking what FORTH actually can do in practice, I would like to ask your help in two areas, namely:

a) FORTH Training Course - We will be organising a Training Course in FORTH in our Hobby Club this autumn. While *Starting FORTH* is an excellent work book for such a course, we really need some more structured material for a 6 to 10 lesson introduction. Any ideas?

b) Actual FORTH Applications - Where can I get a few self-contained applications written in FORTH as examples for our members? The screens given in *FORTH Dimensions* are fine, but I am looking for a few example Games, a Word Processor, a very basic BASIC, that sort of thing, just to show our members what they can expect from FORTH.

Looking forward to hearing from you, and receiving *FORTH Dimensions* regularly.

J. M. Preston
The Netherlands

Congratulations on the success of your club! As for the applications, any that we know of we're publishing here. There were three games described in III/5, a "very basic BASIC" in III/6, I'm including the code for my simplified word processor in this issue, and we expect lots more applications to come. —Editor

Infecting the Mumps

Dear FIG,

I would appreciate receiving some of your publications for book reviews in the publications I edit. In addition to being an editor for the MUMPS Users Group (MUG) Quarterly, I am software/firmware editor for EDP News Service, and manage the Software Digest newsletter. I would like to begin getting information about FORTH into the trade news in the computer industry on a regular basis.

Since December, I have been able to accomplish the following:

— arranged for a full-day tutorial on FORTH at the Annual MUMPS Users Group in Denver;

— arranged for George Shaw's book review of Leo Brodie's text to be reprinted in the MUG Quarterly;

— persuaded Bob Wickizer to include some paragraphs on his use of FORTH as a complement to MUMPS in his radiology system;

— planned for including future articles on FORTH in the MUG Quarterly and in Software Digest.

Henry G. Heffernan
Washington, D.C.

FORTH Dimensions

Published by FORTH Interest Group
Volume IV, No. 3
September/October 1982

Editorial/Production
Leo Brodie

Publisher
Roy C. Martens

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

Letters . . . (cont.)

Freedom and Formatting

Dear FIG,

J. T. Currie has provoked me (as was his stated intention). His statements concerning the formatting of FORTH source lines are not only provocative but outrageous, ignoring many of the diverse user environments in which FORTH is a useful tool and within which the "good" FORTH programmer" may find himself working.

I have used FORTH for many purposes on many machines in many user environments since within a year of its emergence from NRAO. For some types of problems and some user environments it is a marvelous language.

Lately I have been working with high energy physicists and their engineers and technicians. FORTH has been used to: 1) set up and check out an enormous data acquisition system and the apparatus it is connected to, 2) simulate, debug and monitor a very large special purpose processor used in real time data selection, and 3) check out and manage the operation of a multi-processor used for analyzing the data collected.

Our users might be divided into three categories: 1) Novices, who use words already defined and restrict extension of the system to temporary simple lists of existing words; 2) Normals, who extend the system with permanent colon definitions and use those definitions to solve their problems; 3) Nuts, who may reassemble the kernel, write code definitions to connect to new hardware, define new defining words . . .

At any one time, our users are about half novices and half normals with a nut or two hanging around. It's not that the novices and normals are dull slobs who couldn't get beyond that level; they are people who exercise their creativity in other areas and use FORTH as one tool among many to get a job done.

One might wish, indeed I do wish, that everybody learned FORTH in grammar school. Until that happy day arrives, I must work in an environment where most of the users will be non-experts (to say no more). A local prison warden, criticized for the appalling condition of his jail, responded that things wouldn't get better until they got a "better class of prisoners." It makes no more sense for me to yearn for "a better class of technicians." Our programming environment must seek to do the best it can in an imperfect world.

Our engineers and technicians have grown up on Pascal and BASIC and Assembler; the physicists are addicted to FORTRAN. They are used to "lines of code" and conventions of indentation. Novices look at a block of unformatted FORTH code the way I look at Bengali script — incredulity that squiggles speak of truth or beauty or even the proper function of a time digitizer. A screen made up of a hundred or so terse words that look like they were run through a trash compactor on their way to disk is a formidable barrier that I see no point in imposing on our novices. I'm glad that FORTH doesn't insist that code for inner loops be indented three spaces from the next highest level, ecstatic that it allows me to insist on it.

At about the time a user starts advancing from Novice to Normal, I suggest, club in hand, that he adhere to a short list of formatting conventions. Definitions start on the left and are preceded by comments in plain English. Loops and conditionals are indented. Definitions do not (usually) span screens. Screens that change BASE when loaded must save and restore it. Use of a word for different functions in different vocabularies is strictly forbidden; in general, use of non-default vocabularies is forbidden. Words are to have clear mnemonic value whenever possible, even at some expense in brevity. Lots of empty space is left to make later amendment easy.

This does not insure that the code written is good — cleanly factored, concise, fast, sufficiently general, easily used for further extension. That will come with time and training. What it does do is to insure that most of our users and especially our inexperienced and occasional users can use and begin to understand the code as quickly as possible. Formatting is helpful to most of our users, even when the code being studied is excellent. For badly written code (and there is lots of that around in our kind of environment) it is yet more helpful. Unlike the deeper aspects of "good programming practices," the formatting rules can be mastered and applied without serious error within about five minutes of the time they are first explained. That's a pretty low cost for even an incremental improvement in readability.

By now, I have seen about 100 people start FORTH. None has ever complained seriously about formatting requirements. Scores have bemoaned FORTH's tendency to be a "write-only language." As more of our code has been formatted, the moaning of the novices has decreased markedly. To me that's important.

Suppose that FORTH allowed screens to be written backwards or that spaces between words might be omitted unless an ambiguity was thereby created. The latter would certainly result in much more compact code. .ti daer ot nrael yllautneve thgim secivoN But what's the point?

In designing a language, as in designing a government, building in a lot of freedom is good. In each case, it is not always a virtue (or even kind to yourself) to use all of it. It's a virtue of FORTH that it leaves the user community free to decide on formatting conventions appropriate to the task at hand and the needs of the community. There is no forthly imperative to hang yourself just because FORTH gives you enough rope to do so.

Stephen B. Bracker
Fermilab Users Office
Batavia, Illinois

Charles Moore's New Venture

Dear Fig,

I wish to announce a new company in the FORTH arena, Charles H. Moore & Associates. Of the many opportunities available in the growing FORTH community, we plan to specialize in custom applications. At the moment, that means software. But when the FORTH chip becomes available, so will custom hardware.

Moore & Assoc. will be a distributed network of programmers linked to each other through their computers. Thus half of the associates will be computers, and the others humans. Together we will have the flexibility of working at home, at the customer's site, or at the place of our choosing.

Anyone knowing of applications demanding more than one FORTH programmer is invited to contact me: Charles H. Moore & Assoc., Drawer CP-66, Manhatten Beach, CA 90266.

I'll keep you informed of developments.

Charles H. Moore

Forté — A FORTH-Based Operating System

John S. James
Colon Systems

Does FORTH need an operating system? If so, what requirements are most important, and how can a system be designed to provide them? We believe that the answers to these questions are crucial to the future role of FORTH.

FORTH includes its own operating system, and for years most users have been content to run it on a stand-alone basis. The advantages of fast development, transportability among different types of CPUs, interactive flexibility, and reliable and efficient software development are available without any other operating system. But the one major capability which stand-alone FORTH fails to provide is becoming increasingly important — namely, an effective mechanism for reusing software components which may have been developed independently. Forte' is intended to answer this need.

Two Kinds of Transportability

While FORTH solves the problem of moving software between different CPUs, there is another kind of transportability — moving modules between different installations, each with different teams of developers, different conventions, tools and environments. For example, suppose you are developing a fairly large software system and realize that most of the logic has already been coded by others. You would like to tie together a number of FORTH routines from, say, ten different installations, which have developed them independently of each other, without coordination. Naturally you want to treat these modules as "black boxes" and not make any changes to them. Treating each module as a unit avoids the need to learn the internals, and avoids the risk of introducing errors in work which may have been well tested by use in dozens of different projects. In addition, the developers may only want to release running modules, not the source.

This kind of modularity and transportability (which UNIX [*] handles well, a large part of the reason for its success) is not available in stand-alone FORTH. Instead, most FORTH development projects are written from scratch, except for routines which may have been copied (or more usually, adapted) out of listings and retyped, or subroutine packages from a single source, usually provided as part of the system package.

Obstacles to Modularity

What are the obstacles to inter-installation use of pre-developed software modules? These are presented in the order in which they might affect a project.

(A) Incompatible media formats. Available programs can help overcome this problem by transmitting source code serially (see Grotke, G.T., "Transfer of FORTH Screens by Modem, *FORTH Dimensions*, III/5, p.162), but such programs are not widely used, probably because of the other problems which are outlined below.

(B) Only source can be transported — if the developer will release it.

(C) Stand-alone FORTH does not have directories or named files, so screen ranges must be considered when source modules are copied. The source may need to be edited to get it in a reasonable order.

(D) Any word-name conflicts must be resolved. These conflicts will usually involve words internal to the modules, which the developer who is using those modules should be able to ignore.

(E) The final system may not fit in memory. Even if more than 64K is available, stand-alone FORTH is not automatically set up to use it.

(F) If the system does not fit, a special overaly technique may need to be developed, and the modules may have to be changed to accommodate it. Perhaps the modules can be rewritten to use extended memory for data.

(G) Due to these problems, there is no library of modules in widespread use. Each project is developed from scratch.

(H) Even within the same installation, it is difficult to use unchanged source-program segments for different application systems. New projects often must recode or at least modify previous work.

FORTH otherwise provides such a powerful environment for fast development of correct programs that the major penalty resulting from these difficulties is often overlooked or found acceptable. The opportunities which are lost because the FORTH environment does not easily produce interchangeable libraries of large program segments are not missed, because they have not been experienced. But in the future, FORTH must increasingly compete with systems such as UNIX, which do support this function. (For example, when UNIX is distributed, hundreds of modules written by many different users at different installations go with it.)

What's Needed?

The most important requirements for modularity are a file system, and a program library. The entire operating system can be smaller and simpler than most, since FORTH itself provides many of these functions. Several simplifying assumptions are used in the Forte' operating system:

(A) We are assuming a single-user system. Multitasking will be provided, but not inter-task protection. Many of the micros on which FORTH is customarily used do not have facilities for real inter-user protection: instead, protection should be provided at the local-network level. Simple multitask-

Continued

ing allows utilities such as printer or communication spoolers to be used by a single developer or operator.

(B) The file system provides files of FORTH blocks, so that existing programs which were not written for a file system can be run without modification. Normal source code and conventional screen editors can be used. But under the operating system, all disk access is invisibly constrained within those files which the user is authorized to open. There is no error message for attempting to access unauthorized data; instead, there is no such thing as 'BLOCK' accessing outside of open files. Effective data protection becomes especially important as non-removable hard disks replace floppies, because in many installations a single micro will serve several users who all keep data on line.

(C) Not only is 'BLOCK' defined on top of the file system; the file system is defined on top of 'BLOCK'. Besides the obvious advantage of transportability of the operating system, this design also isolates any complexity below 'BLOCK' (such as support of different types of devices, and support of the use of non-error-free media) from the rest of the system. The operating system only sees sets of block-number ranges; since it can be configured for any ranges desired, it can organize only parts of disks if desired.

The File System

Allocation is dynamic and non-contiguous. All files are created with length zero, and any access to a block number allocates a block of blanks to it. If contiguous allocation is desired it can easily be forced.

Hierarchical directories, and UNIX-style pathnames, are provided. Both require low overhead, and are increasingly important for large disks. A branch of the file hierarchy can be a dismountable volume.

All files start with a header block, which contains the information necessary to restore the file. Therefore files can be recovered even if directories are destroyed.

A special file type important enough to be provided separately is the automatically-sorted file, e.g., using B-trees. Many applications need to keep online files of records sorted by fields such as part number or customer name; the data should always be current, and the user should not have to wait for batch sorts.

The Program Library

We have chosen a simplified library mechanism, to keep down overhead, by taking advantage of facilities already available in FORTH.

The goal is to support a library of precompiled routines, called "modules," which can be called from other modules, from a main program, or from the keyboard, in exactly the same way.

Each module lives in a file in the hierarchical directory. When run, the module may be identified by name or pathname. If the module is not found by a search starting in the "current" directory, other directories in a search vector may be checked automatically. Usually the search vector specifies program libraries.

All modules have the same starting address in memory. If one calls another, the calling module is saved and restored. Modules may or may not include headers. Those with headers link to the resident ("root") dictionary.

Modules normally communicate with each other by the FORTH stack. Module calls can be used like other words in ordinary FORTH definitions, as long as the modules themselves are available in a program library when needed during execution.

Summary

Conventional FORTH systems do not effectively support libraries of subroutines larger than single definitions but smaller than complete programs. As a result, many software projects cannot take advantage of previous work.

A file system and a library mechanism for precompiled routines are the essentials for extending FORTH's usefulness in the new environment of larger micros, larger projects, and modular software development. □

*UNIX is a trademark of Bell Laboratories.

John S. James is working for Colon Systems in San Jose, California, on the development of the Forte' Operating System.

A Standardized Microcomputer Operating System Interface?

Gary Feierbach

The Institute of Electrical and Electronic Engineers (IEEE) has an active committee PI55 dubbed MOSI which is chartered to define a set of programmatic interfaces between microprocessor based operating systems and that software which utilizes operating system functions. Today we have what essentially amounts to chaos in this area with the exceptions of a few proprietary vendor packages which have become defacto standard like CP/M and UNIX.

MOSI came out with a draft document (Rev 3.0) which is in the process of being revised and is soon to be released (Rev 4.0 in August or September) as a proposed standard. The draft document reveals operating system or modules from 1 to 6. These capability modules span very simple environments as in the case of dedicated controllers to complex environments involving memory, clock, data and process management.

The current draft (Rev 3.0) is quite bulky covering 161 pages and Rev 4.0 promises to be under 100 pages. The MOSI committee is also bulky consisting of over 268 members (it was 600). One would think such an effort was doomed from the start to produce an elephantine monstrosity. A monstrosity it is not. As a matter of fact the result, to this point, looks quite plausible. A good part of the bulk is due to examples of using the O/S primatives from C, BASIC and COBOL. They need someone to provide examples from a FORTH environment.

Furthermore, if anyone is up to the challenge, implementing a MOSI O/S (in FORTH of course) would go a long way in helping clean up problems in the proposed standard. Who knows, it may replace CP/M or UNIX or both (or neither).

To get on the MOSI committee contact Jack Cowan, Intel Corporation (M/S DV2-210), 2402 West Beardsley Road, Phoenix, AZ 85027.

The FORTH community should be involved in this effort. □

A FORTH Based File Handling System

Dr. Donald Delwood
Columbia, Missouri

Introduction

This article will discuss the topic of file handling that has been developed under the operating systems, TR11 and RSX11, with principles applying to many others including CP/M.

Fig-FORTH inspired versions typically use one large file in which to store source screens of FORTH code. Even casual glances through *FORTH Dimensions* reveal screens labeled into the thousands. Finding an application usually involves either wide searches of screen ranges using the INDEX word or the use of documentors or documentor screens (hopefully meticulously maintained). Furthermore, this approach requires much "housekeeping" as applications grow crowded requiring massive screen-moves or fragmented multi-linked screens. If several screens of code are to be distributed, the whole large file must be given out, or other methods such as hardcopies or telephone transfers resorted to. Furthermore, large files are just less convenient to manipulate on the smaller floppy disk based systems.

The advantages of having an extensive files system under FORTH would be many. Each application or related applications could have a file. This is hardly novel, in fact it is the "usual" way of doing things in many other programming systems. A clearly named file may be loaded by an even clearer FORTH word. Specific applications may be passed among users with ease. Users of the same system at different times can each have their own file(s) or environment. Generally useful words created during an application development may be easily and clearly placed in the generally-useful-word file rather than linking another screen to the already widely disseminated batch of screens in the one-big-file system. In other words, clear, easy to use libraries would evolve.

Operating systems revel in manipulating files. It seems sometimes that their only purpose to exist is to help with CREATEing, DELETEing, COPYing, PROTECTing, MERGEing, EXTENDing, DATEing, LISTing, and DIRECTORYing files. Since many FORTH users run under an operating system rather than stand alone, it makes good sense to tap into this resource of well debugged and sometimes even documented code.

I will use examples from RT11, the single-user operating system by Digital Equipment Corp. for their PDP-11 computers. It is quite mature, with a fine repertoire of useful monitor calls. However, the same approach was used in development for RSX11 and is being used for CP/M.

From the top down

A useful file system would use the system's specific naming convention, be able to either find a pre-existing file or create a new one, and close the file after access. That's all that is really needed. Optionally, it might be useful to delete, rename, as well as those other utilities listed above, but certainly that can be done later by the operating system outside of FORTH.

The mechanics of file handling in the three operating systems RT11, RSX11 and CP/M are very similar. A block of memory called the File Descriptor Block (FDB) is loaded with the file name, device on which it resides, and other useful information according to a system specified convention. Then certain action words or verbs, which will be introduced shortly, associate a channel (a logical I/O pathway) with the file specified in the FCB. I/O from then on can be specified simply by indicating which channel is to be used in subsequent reads and writes. Thus, the minimal system requires a word specifying the address of the file descriptor block, **FDB**; a word that somehow packs the file specifications into the FDB, **FILE**; a variable containing the currently active channel, **CHAN**; and file active verbs, **ENTER**, **LOOKUP** and **CLOSE**. See definitions below.

- **FDB** (-- addr) FDB returns the starting address of the system specific file descriptor block.
- **FILE** (--) FILE can have many forms as long as it fills the FDB with the proper information. This article will describe a FILE that expects a string on top of the string stack that is a standard RT11 file descriptor. It then parses it, converts it to RAD50 and shoves the letters into the proper position in the FDB according to RT11 requirements.
- **CHAN** (-- addr) CHAN is a variable that holds the value of the currently active channel.

Figure 1a

```
CLOSE                          ( close current file )
" DK:EDITOR.FTH"               ( file specification string )
FILE                           ( pack string into FDB )
LOOKUP                         ( try to find and open the file )
IF ABORT" File Lookup Error" THEN     ( error handling )
1 LIST                         ( get to work )
```

Figure 1b

```
CLOSE                          ( close current file )
" DK:TEST.DAT" FILE            ( fill FDB with file name TEST.DAT )
LOOKUP IF                      ( does such a file exist? )
          20 ENTER IF          ( nope, so make one 20 screens long )
                      ABORT    ( oops, big-time error )
                    THEN
       THEN
( perform any output )
CLOSE                ( close file )
```

APPENDIX A, RT11 Files Handling Source

This was added to FORTH in assembly language simply because the RT11 monitor calls are basically macros and end up looking much prettier if left unexpanded.

Note: The HEAD and NEXT macros used is exactly compatible with the FORTH PDP-11 listing distributed by FIG.

Note: Although CHAN is included, the read/write routines need to be taught to use CHAN as well. As mentioned in the text, either don't use CHAN for now, or contact me.

```
    HEAD    204,CHAN,240,CHANL,DOVAR                ; ***** CHAN
    .WORD   0

    HEAD    203,FDB,302,XFDB,DOCON                  ; ***** FDB
    .WORD   FSB

    HEAD    206,LOOKUP,240,LOKUP            ; ***** LOOKUP
    MOV     CHANL+2,R1      ; GET CHANNEL FROM  VARIABLE CHAN
    CLR     -(S)            ; ASSUME NO ERRORS
    .LOOKUP #EMTBLK,R1,#FDB
    BCC     1$              ; DONE IF NO ERROR
    INC     (S)             ; PUT TRUE ON STACK FOR ERROR
1$:         NEXT

    HEAD    205,ENTER,322,ENTER             ; ***** ENTER
    MOV     CHANL+2,R1      ; MOVE THE CHANNEL TO R1
    ASL     (S)             ; CONVERT FORTH-BLKS TO DEC-BLKS (2*)
    .ENTER  #EMTBLK,R1,#FDB,(S)
    CLR     (S)             ; CLEAR ERROR FLAG TO FALSE
    BCC     1$              ; DONE IF NO ERROR
    INC     (S)             ; PUT TRUE ON STACK FOR ERROR

1$:         NEXT

    HEAD    205,CLOSE,305,CLOSE             ; ***** CLOSE
    MOV     CHANL+2,R1      ; GET CHANNEL FROM USER VARIABLE CHAN
    .CLOSE  R1              ; CLOSE CHANNEL
1$:         NEXT

EMTBLK:     .BLKW   6               ; EMT BLOCK
FDB:        .RAD50  /DK FORTH DAT/  ; FILE NAME BLOCK DEFAULTS DK:FORTH.DAT
```

- **ENTER** (n -- tf) ENTER will create a new file of maximum n FORTH blocks in length. Returns a flag, tf, which is 0 if successful or 1 if unsuccessful.
- **LOOKUP** (-- tf) LOOKUP opens an existing file and returns an error flag similar to ENTER.
- **CLOSE** (--) CLOSE closes the currently open file.

A listing of the code for the implementation (as well as some interesting developments under the DEC o/s) is in the appendix.

An example of typical code to open a file appears in Figure 1a. A definition to open and write to a data file appears in Figure 1b.

Note: The RT11 file specification is: device (DEV) which is a logical name assigned to the random access mass storage unit, followed by a maximum 6 letter file name, followed by an optional 3 letter extension, e.g., DEV:FILNAM.EXT.

Text continued on page 22
Appendix B is on next page

APPENDIX B
Development of the word FILE and misc.

Note 1: I'm sorry, but most DEC users must deal with RAD50.

Note 2: This code uses the word **WITHIN** which is defined as follows:
: WITHIN (n low high -- f) (low <= n < high) >R 1- OVER < SWAP R> < AND ;

```
****** DK:FILES.FTH *******        4     ************************

( Files routines - RAD50 conversion )
octal
: >r50 ( ascii -- rad50 )
  dup  40 =           if drop 0  else
  dup  44 =           if drop 33 else
  dup  56 =           if drop 34 else
  dup  60  72 within if  22 -    else
  dup 101 133 within if 100 -    else
  ." Illegal RADIX-50 character: " 42 emit emit 42 emit cr
  then then then then then ;
decimal

( basic routine to convert an ASCII character to RAD50
  gives error message if try to convert an illegal character
)



****** DK:FILES.FTH *******        5     ************************

( Files routines - RAD50 conversion )
octal
: (r50>) ( rad50 -- ascii )
  dup    0=           if drop 40 else
  dup 33 =            if drop 44 else
  dup 34 =            if drop 56 else
  dup 36 50 within    if 22 + else
  dup  1 33 within    if 100 +
  then then then then then ;
( basic routine to convert RAD50 to ASCII )

: r50> ( n -- asc3 asc2 asc1 )
  0 50 u/mod 0 50 u/mod
  (r50>) rot (r50>) rot (r50>) rot ;
( converts a 1 word integer in RAD50 to 3 ASCII characters )
decimal



****** DK:FILES.FTH *******        6     ************************

( Files routines - RAD50 conversion )
octal
: >rad50 ( addr -- n )
   dup 3 + swap do i c@ >r50 loop
   swap rot 50 * + 50 * + ;
( converts 3 characters, i.e. bytes starting  at addr )
( into single 1 word integer RAD50 equivalent )
decimal

: rad50> ( n addr -- )
  >r r50> r@ c! r@ 1+ c! r> 2+ c! ;
( converts RAD-50 n into 3 characters, bytes, starting at addr )
```

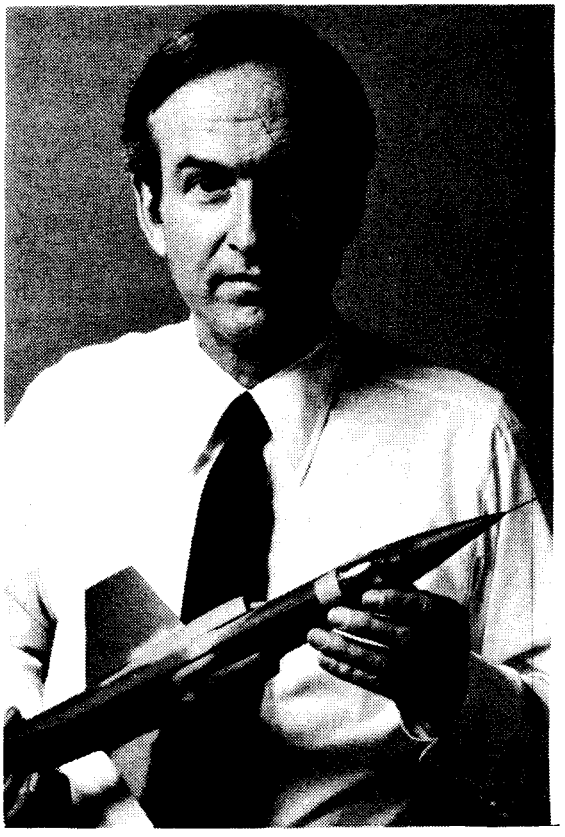
```
****** DK:FILES.FTH *******       7     ************************

( Files routines - file strins handlins )

: $r50> ( n -- ) r50> emit emit emit ;
( prints the 3 characters stored as RAD50 integer n )

: $filnam ( addr -- )
  dup    @ $r50>                  ( print FIL_______ )
  dup 2+ @ $r50> 46 emit ( . )    (       ___NAM.___ )
      4+ @ $r50> ;                (       _______EXT )

: $filspec ( addr -- )
  dup    @ $r50> 58 emit ( : )    ( print DEV:     )
      2+   $filnam ;               ( print FILNAM.EXT )

( prints file specifications stored in 4 word block at addr )



****** DK:FILES.FTH *******       8     ************************

( Files routines - file strins handlins )

: $filefill ( addr1 addr2 )
  14 over c!                      ( store lensth byte )
  1+ 4 0 do
     over i 2* + @ over rad50> 3 + loop
  2drop ;

( if 4 word file specifications besins at addr1 )
( will make a TYPEable strins at add2 )




****** DK:FILES.FTH *******       9     ************************

( Files routines - file array )

: ()filblk <builds 8 * allot does> chan @ 8 * + ;
16 ()filblk chanblk
( create a strins array to hold 16 file block specifations )


****** DK:FILES.FTH *******      10     ************************

( Files routines - file block packins )

: file ( $filspec -- )  ( puts file spec into filblk )
  $sp @ 2+                        ( jump over letter count )
  filblk dup 8 + swap do          ( set up index )
       dup >rad50 i !
       3 + 2 /loop                ( increment addr, and loop )
  $drop drop                      ( drop strins and filblk )
  filblk chanblk 4 move ;         ( store file name in array )

( primitive routine! - takes a strins in the form: )
(              " DEVFILNAMEXT"                     )
( on the strins stack, then packs it in proper format )
( into file specification block )
```

TEST-FLY A $20 MILLION JET ON AN APPLE? YES. WITH MICROSPEED.

At the Bethesda Naval Research Center, they've discovered the power of MicroSPEED. The Navy's engineers use this remarkable hardware/software combination to "fly" an advanced fighter aircraft in *real time*—even making vertical landings on a simulated carrier deck. A "crash" is merely another learning experience, and an opportunity to modify the research aircraft—inside the Apple—to improve tomorrow's combat planes.

Surprised that such a sophisticated task is possible on the Apple? So were the Navy's officials, and many others who have discovered...

THE MICROSPEED DIFFERENCE This extraordinary Language System exploits the real potential of the microcomputer for the first time. The difference between MicroSPEED and other programming languages is that with MicroSPEED, there is virtually *no limit* to what you can achieve. It may well be the ultimate language for the Apple II and III (and soon the IBM Personal Computer). MicroSPEED literally combines the performance of a *minicomputer* with an exhaustive set of user-friendly capabilities:



hardware math processing, fast hi-res graphics and text, turtle graphics, print formating, two text editors, unlimited data types, and incredible FORTH extensibility—all at speeds up to 100 times faster than Basic.

USER-FRIENDLY, EASY-TO-LEARN Starting with simple commands that are comfortable even for non-programmers, MicroSPEED extends and builds, allowing you to create your own tailored application languages. The capability of your computer will grow exponentially, as you work in an active partnership with the machine, exploring and developing new problem-solving facilities—creating, correcting, refining your increasingly powerful system.

DEMANDING JOBS AT LOW COST Developed by a team of standout computer professionals, MicroSPEED has been put to the test in fields as diverse as medicine, the stock market, oceanography, and the arts. In even the most challenging applications, MicroSPEED users have been unanimous in their praise of the System and manual. Typical comments are:

"Very high marks,"
Thomas Tosch Ph.D., Tosch Information Management.

"The more I use MicroSPEED, the more I love it,"
Prof. James L. Hockenhull, University of Washington.

"Great!...A joy to use,"
Henry Harris, Mission Designer, Cal Tech's Jet Propulsion Lab.

"If you plan to use the Apple or IBM Personal Computer for <u>any</u> demanding task, then we built MicroSPEED for <u>you</u>,"
Sam Cottrell, President of Applied Analytics.

MicroSPEED requires the Apple or IBM Personal Computer with single disk. MicroSPEED II includes 2 MHz math processor. Micro SPEED II+ includes 4 MHz math processor.

Applied Analytics Incorporated
8910 Brookridge Drive
Upper Marlboro, Maryland 20772 (301) 627-6650

I'm interested! My computer is:_____

Please send me:

_____MicroSPEED II, $495.00 _____160 Page Manual, $15.00
_____MicroSPEED II+, $645.00 _____Detailed Information

Name:_____

Company_____

Address_____

City_____State_____Zip_____Phone No.: ( )_____

Use this coupon to order, or for more information.

MicroSPEED μSPEED

APPLE IS A TRADEMARK OF APPLE COMPUTER INC.

Combining 79-Standard FORTH with Existing Microcomputer Operating Systems

John Arkley
The Software Works
Palo Alto, California

During our recent conversion of Software Works FORTH into 79-Standard FORTH, most of the problems of building a FORTH on top of several existing microcomputer operating systems were tackled and solved. This article briefly discusses the advantages and disadvantages of our approach.

Building FORTH upon a wide range of different operating systems requires two virtual models for FORTH input/output, one for character or device I/O and one for disk I/O. Conveniently, 79-Standard FORTH defines these via the words **KEY**, **EMIT**, **EXPECT**, and **TYPE** for character I/O and **BLOCK**, **BUFFER**, and **SAVE-BUFFERS**, and **EMPTY-BUFFERS** for disk I/O.

79-Standard FORTH fails to address the problems of directories, directory management, file creation, device definition, device assignment, special driver linkages, keyboard mapping, function keys, and the like. The philosophical reason is that no one operating system will ever do what is desired, and it usually hinders rather than supports the programming language system.

Advantages

There are a number of advantages to using the native operating system, even though doing so increases the size of the FORTH system.

A primary advantage is the ability of such a FORTH to co-exist on the same mass-storage volumes with other application software and languages. Now that we're seeing cheap high-density storage, the usual FORTH design (which can be summarized by the phrase 'I own the whole disk') is an unacceptable one. The massive microcompuer market requires a different FORTH approach, one which supports sharing resources with other software systems.

Another advantage is the easier acceptance of FORTH by non-FORTH programmers still undergoing the FORTHification process.

None of the common micro computer operating systems support files that the system will span across multiple disk volumes. This problem can be solved in the interface between FORTH and the host system, thus reducing the applications programmer's concern for disk capacities. FORTH can circumvent deficiencies in these operating systems.

A similar scenario applies to character I/O. The best example of this is the two and a half character devices in CP/M. FORTH can easily interface I/O through a vector table that allows patching in device drivers for 8 or 16 devices.

Another example is found in device paralleling, a means of directing the output stream to multiple output devices without having to put special-case code in an application.

Disadvantages

Using an existing operating system is not without some costs, some of which affect performance. A good example can be seen in the difference between character at a time I/O with **EMIT** and a line at a time I/O with **TYPE**. If the overhead for an output request is very high, software that only uses **EMIT** could easily run 10 times slower than if whole lines were **TYPE**ed for most of its output. Disk I/O speed can also suffer if the operating system is slow at random access reads or writes.

The final disadvantage is encountered with operating systems that are entirely useless for FORTH or too complex to support the simple needs of FORTH. The problem is solved by using the directory format and disk allocation mechanism, and implementing a subset operating system from scratch that is sufficient to support FORTH. This problem arises when the operating system is inside a disk BASIC, ala MICROSOFT disk BASICS, or such first-generation systems like APPLE DOS 3.X which is very slow for large random access files due to its linear index structure and its unusual post-basic interface design.

Implementation Considerations

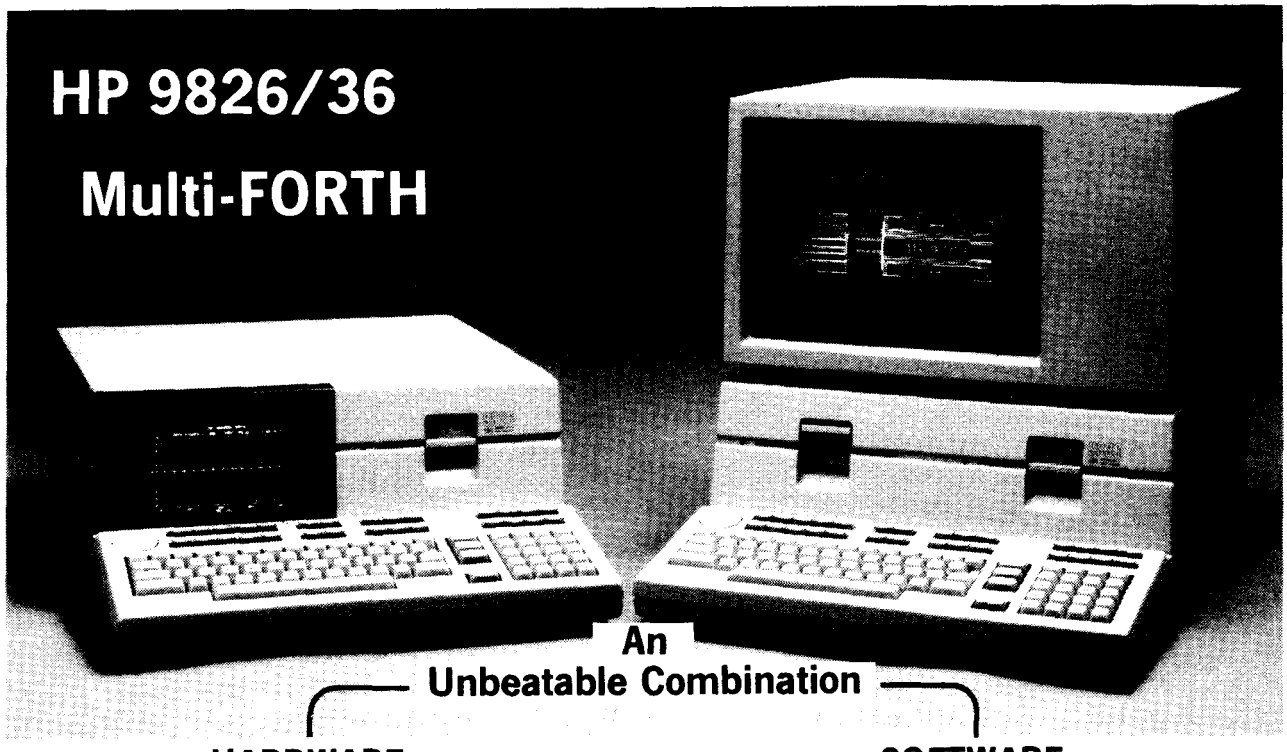
The actual cost of doing an operating system from scratch, as we have done for the APPLE II, is not as large as one would think. It only took 3 weeks to create the 6502 FORTH, the operating system, and a new disk driver for the APPLE disk controller. This is largely due to two factors: most of FORTH can be written in itself, and I/O models that are simple but elegant, and therefore easy to duplicate, were used.

Most micro systems have the needed random access facilities and can have their disk file I/O interfaces mapped directly to the 79-Standard FORTH 1024-byte definition of a block. This does result in not being able to use all possible files that the host operating system could create, since 79-Standard FORTH's disk I/O design doesn't provide for partial block input or output. One possible solution for this problem is to create a character driver for **KEY** and **EMIT** to provide sequential character access to a disk file.

The results of combining microcomputer operating systems with 79-Standard FORTH produces an environment that remains entirely FORTH and at the same time allows real transportability that is acceptable to the large majority of existing microcomputer users. The performance and size costs to the resultant FORTH will, no doubt, be hotly argued from different views, but the bottom line, selling software products in volume, outweighs all possible technical perspectives. We expect to see a much larger acceptance of FORTH applications software as the result of this approach, particularly from the hardware manufacturers like OSBORNE and SANYO. □

Now Available On

HEWLETT PACKARD DESKTOP COMPUTERS



An Unbeatable Combination

HARDWARE

The HP 9826A and 9836A are two of Hewlett-Packard's newest and most powerful desktop computers. Each is based on the Motorola MC68000 microprocessor. Both machines have full graphics capability and up to 2 full megabytes of user read/write memory. Both operate on 5¼" flexible disc drives (the 9836A has two) which feature 264K bytes of mass storage. While the 9826A has an integral 7" (178mm) CRT which makes it useful for computer-aided testing (CAT) and control, the 9836A has a full 12.2" (310mm) CRT which makes it ideal for computer-aided engineering (CAE) applications. Each model features the following:

- Seven levels of prioritized interrupt
- Memory-mapped I/O
- Built-in HP-IB interface
- Standard ASCII keyboard with numeric keypad and international language options
- Ten (20 with shift) user-definable soft keys with soft labels
- Rotary-control knob for cursor control, interrupt generation and analog simulations
- System clock and three timers
- Powerfail recovery option for protection against power lapses
- Seven additional interface cards
  - — DMA controller (up to 2.4 mb/sec)
  - — 8/16 bit bi-directional parallel
  - — Additional HPIB interface
  - — Serial RS232/449
  - — BCD
  - — Color video(RGB) 3 planes 512 x 512 8 color

SOFTWARE

HP 9826/36 Multi-FORTH
HP PRODUCT # 97030JA

Multi-FORTH was developed in 1979 by Creative Solutions, Inc. The standard product has been substantially modified to take full advantage of the 9826/36 hardware features.

Multi-FORTH features

- 79 standard programming environment
- Multitasking
- Full screen editor
- In-line structured assembler
- I/O and graphics extensions
- Loadable H.P. floating point (IEEE format)
- Extensive user manuals and documentation

Optional Features:

- Meta compiler
- Multi user
- Data access methods library

This product is part of HP PLUS — a program for locating user software. It has been developed by an independent software supplier to run on HP computer systems. It is eligible for HP PLUS as determined by references from satisfied end users. Support services are available only through the software supplier. Hewlett-Packard's responsibilities are described in the Responsibilities Statement below.

Responsibilities Statement

HP PLUS software was developed by an independent software supplier for operation on HP computer systems. The supplier is solely responsible for its software and support services. HP is not the manufacturer or developer of such software or support. HP disclaims any and all liabilities for and makes no warranties, expressed or implied, with respect to this software. Distribution of this product or information concerning this product does not constitute endorsement of the product, the supplier, or support services. **The customer is responsible for selection of the software it purchases.**

For more information, please write

Marvel Ross, Hewlett Packard Company

3404 East Harmony Road, Ft. Collins, CO. 80525

Checksum for Hand-Entered Source Screens

Klaxon Suralis
and
Leo Brodie

For program exchange, the medium of hard copy is cheap, convenient, and machine-independent. Its primary disadvantages are the time required for hand-typing the source code and the possibility of human error in the process. Even if the screens **LOAD** without error messages, some errors (omitted or transposed words, frinstance) may pass undetected until run-time, when the system crashes mysteriously.

This program can ameliorate the latter disadvantage, although it does nothing about the former. Given a screen number on the stack, **VERIFY** calculates a 16-bit CRC value which may be printed above each screen's listing. The check value is NOT written into any block or buffer.

On the receiving end, **VER** is used to compute the check sum of the screen as typed in. If the number differs from that given on paper, something is definitely wrong; if the values are equal, chances are very good that the program compiles as listed.

We've chosen to express the checksum as an unsigned number; for the sake of standardization we suggest you do the same. By the way, **VERIFY** may take a second or two — be patient.

Note that comments are ignored in the CRC generation. Also, the number of spaces between words makes no difference. You may rearrange source words and change/omit/add comments without affecting the value returned by **VER** and **VERIFY**. But a missing space will, as you'd expect, change the checksum.

The program runs on fig-FORTH provided you make the following redefinitions before loading this application:

```
: WORD   WORD  HERE ;
: >IN    IN ;
: NOT    0= ;
```

For some systems you may need to predefine **32 CONSTANT BL**.

The program will NOT work, however, on FIG systems which don't have disk buffers consisting of 1024 bytes of contiguous memory. Sorry.

The program has been designed for transportability, at some expense of speed. It depends on **WORD** to parse groups of letters separated by spaces. Since certain details of **WORD**'s operation vary from version to version, we chose an algorithm that would cover all bases.

The main trick was the test to decide when to end the outer loop. On 79-Standard systems, when **WORD** hits the end of the block it returns a count of zero; but on FIG systems, it returns a count of one. Therefore, a test for zero count isn't transportable.

The algorithm we chose was this: keep **DISPOS**ing word-strings until the next string parsed by **WORD** has a count less than two (which covers zero and one) and consists of an ASCII character less than 33 (which covers blank and null). Thus, in a FIG system, if the count is one, but the character is blank or null, we're at the end of the block. This test is made in the word **MORE**.

If you feel a need to optimize this test for your system, make sure that a) you get the same results as published here, and b) you get the same result whether or not the final word in a block occupies the last bytes of the block.

Further cautions:

The program uses FORTH-79 **WORD**. It can't handle comments longer than 255 characters.

The program uses unsigned addresses as **LOOP** parameters. As long as **WORD**'s buffer doesn't cross the 32K boundary, everything's okay; otherwise you'll have to alter **DISPOSE**.

Finally, we can't guarantee that the CRC algorithm used in **ACCUMULATE** is the best on land and sea. It's adapted from a book, and seems to work. We'd like to hear about flaws or improvements.

We hope that this routine will become a standard. We'd like to see FORTH vendors include some compatible version of it in their systems. Perhaps embedded as an option within **LIST** or **TRIAD**.

Good luck.

```
Screen # 129            crc ver = 56445
  0 ( Checksum for hand-entered source screens)
  1 : ACCUMULATE  ( oldcrc\char -- newcrc )
  2    256 * XOR  8 0 DO  DUP 0< IF  16386 XOR  DUP +  1+
  3                              ELSE  DUP +  THEN  LOOP ;
  4 : DISPOSE  ( crcvalue\adr\len -- newcrcvalue )
  5    OVER DUP  C@  40 =  SWAP 1+ C@ BL = AND  OVER 1 = AND
  6    IF  ( comment; skip it)  2DROP  41 WORD  DROP
  7    ELSE  1+ OVER + SWAP  DO I C@ ACCUMULATE  LOOP
  8    THEN ;   ( careful; LOOPs on addresses)
  9 : MORE ( -- adr f) BL WORD DUP C@ 2 < OVER 1+ C@ 33 < AND NOT ;
 10
 11 : VERIFY  ( scr# -- crcvalue) BLK @ >R  >IN @ >R  BLK ! 0 >IN !
 12    0  BEGIN  MORE WHILE  BL OVER COUNT + C!  COUNT DISPOSE
 13    REPEAT DROP  R> >IN !  R> BLK ! ;
 14 : VER   SCR @  VERIFY  U. ;
 15


Screen # 130            crc ver = 5038
  0 ( Test screen)
  1      For program exchange, the medium of hard copy is cheap,
  2 convenient, and machine-independent. Its primary disadvantages
  3 are the time required for hand-typing the source code and the
  4 possibility of human error in the process. Even if the screens
  5 LOAD without error messages, some errors may pass undetected
  6 until run-time, when the system crashes mysteriously.
```

Q T F
Quick Text Formatter — Part I

Leo Brodie
Chatsworth, California

Like many other writers, I've become addicted to word processors. Trying to write any other way seems unimaginably tedious — like plucking your own chickens for dinner.

A year ago, when I left my full-time job and word processor, I found myself writing at home, trying to cope with a medieval device called the electric typewriter. It was a nightmare; my mistakes actually appeared on paper, the instant I made them! Corrections of a sort could be made, using a bizarre strip of tape coated with a flaky white chalk, or a pungent paint which took three days to dry. (I have a dim recollection of using the electric typewriter and its awkward accouterments years before, only they hadn't seemed so archaic. Must have been a previous lifetime.)

Well, as soon as I got my IBM Personal Computer and a version of FORTH, I proceeded to write a quick, scaled-down word processor, using the same syntax as that used at FORTH, Inc. The formatter portion occupied three screens, and the editor portion nine screens — not terribly sophisticated, but at least the flaky tape and smelly fluid were back in the drawer.

This two-part article features that simplified application, which I call QTF, the Quick Text Formatter. For those of you who are running FORTH but don't have an off-the-shelf word processor (whether because you don't write enough to justify it, or because you can't afford it this month), you may find this application quite attractive.

The basic approach takes some getting used to, but it has some definite advantages aside from simplicity. In fact, over the past several months I've extended this approach to include many features I need in producing technical manuals. And the enhanced version will most likely become a product.

Another reason I'm publishing the QTF here is this: if any authors of future *FORTH Dimensions* articles were to use this application, it would be extremely easy to transfer the text using any routine for transfering FORTH screens. Eventually I'll be able to phone this directly to our typesetter, saving everyone a bit of work.

The final reason I'm publishing the QTF is because we need more published FORTH applications. This one represents an interesting solution to a real problem.

The Approach

As I said, the approach takes some getting used to. The main surprise is that you don't edit an image of the final output as it will be printed. Instead you edit a string of formatting commands and text into a FORTH block. The formatting occurs when you **LOAD** the block. That's the key to the application's simplicity: the formatter uses FORTH's interpreter. As an added boon, the word processor becomes fully extensible.

Of course if you don't like being limited to 1K blocks of text, you can design the application to use block-based files. But if you use the present system carefully, you'll find it's actually convenient to compose your text in "modules," each module being a block containing one or two paragraphs. To edit a paragraph, you edit the module — you don't have to scroll through a long document to find the paragraph you're looking for. It's a "random access" approach to text editing.

The Syntax

Let's take a quick look at the syntax for the formatting commands. The main formatting word is **[** (left-square-bracket). This word indicates the beginning of text, delimited by right-square-backet. You can think of it as a glorified **."** that does carriage returns at the right margin, and formfeeds at the bottom of the page.

(Yes, I know this conflicts with FORTH's own square bracket, but it's just too nice a word not to use here. If you think it's a problem, either use a separate vocabulary, or save this definition for last, and only load this application when you're formatting text.)

For example:

[This string, which would appear in your source formatting block, would format this paragraph making the appropriate line breaks at the right margin. **]**

Obviously, you can't cross block boundaries while inside text.

The word **pp** is a formatting command for starting a new paragraph; it causes the formatter to do two carriage returns. For instance, the source:

[This is paragraph A. **] pp [** This is paragraph B. **]**

would produce

This is paragraph A.

This is paragraph B.

Essentially, in creating a document you write a "program" to format the document, using this specialized language.

The word **cr** does a single carriage return. Like **pp** it repositions the cursor (or printhead) at the left margin (set by the constant **LMARGIN**).

The words **tab**, **indent**, and **hang-ind** allow you to get away from the left margin in very convenient ways.

You'll find all these formatting commands in the accompanying glossary (Figure 1).

What about editing?

If you've looked at the code for the formatting commands (Figure 3), you may be saying "There's nothing here! This is useless."

Well, you're half right. The other half of this application, which I'll feature in Part II, is the text editor that you need to edit your text source. It's impractical to use ordinary FORTH line editors, because they treat a block like 16 lines of 64 characters. If you insert a character in the middle of line 5, any text at the end of line 5 "falls off." Instead we want it to wrap around to line 6. Also, we don't want to be editing inside a rectangle where

Figure 1

```
                    Quick Text Formatter
                       User's Glossary

start           sets everything up.  Use at beginning.
[               begins setting text.  There must be a space
                   between the "[" and the first character to
                   be printed.  For example:
                     [ This is your text.]
]               ends setting text.  Doesn't need a space preceeding.
{               same as [, but allows you to print strings with
                   a ] inside.
}               goes with {
cr              does one carriage return.  Returns to normal
                   tab
pp              begins a paragraph (does two carriage returns).
crs             does specified number of crs
newpage         causes text to begin at the top of a new page,
                   leaving the rest of current page blank.
tab             tabs over to the position specified, relative
                   to the left margin -- for the current line
                   only.  For example, "5 tab" tabs in 5 spaces
                   from the left margin.  But if the text should
                   continue to the next line below, it will
                   begin at the normal left margin.
indent          does a tab to position specified, and causes
                   subsequent lines, up to the delimiter, to
                   be indented to the same tab
hang-ind        does a tab to position specified and causes
                   subsequent lines, up to the delimiter, to
                   be indented three more spaces to the right.
                   This paragraph uses this format, which typesetters
                   call "hanging indent."
center[         like [, begins a string of text, but centers
                   it on the page.
r[              takes a number "n" off the stack.  Like [, begins
                   a string of text, but right-justifies it
                   in a field "n" wide.
sub             "subitem".  Does a  cr  and a  5 hang-ind.
                   Convenient for making certain kinds of lists,
                   including outlines.
subsub          "sub-subitem".  Does  a cr  and a  10 hang-ind.


To produce the following list:

                                Income          Expenses

     1981                       $10.32           $15.12
     1982                  $328,543.21      $408,964.86

we first defined:

: year    cr   5 tab ;
: income    20 tab  14 ;
: expenses    40 tab  14 ;

then wrote

pp income r[ Income]  expenses r[ Expenses]  cr  year [ 1981]
income r[ $10.32]  expenses r[ $15.12]     year [ 1982]  income
r[ $328,543.21]  expenses r[ $408,964.86]
```

Figure 2

```
73

( Formatter Glossary)                  : means  15 hang-ind ;
start  center[ Quick Text Formatter] cr  center[ User's Glossary]
pp [ start] means [ sets everything up.  Use at beginning.] cr
{ [} means [ begins setting text.  There must be a space between
the "[" and the first character to be printed.  For example:]
cr 20 tab { [ This is your text.]} cr { ]} means [ ends setting
text.  Doesn't need a space preceeding.]  cr [ {] means { same
as [, but allows you to print strings with a ] inside.}  cr [
}] means [ goes with {]     -->
```

words may appear to be cut in half if they happen to straddle a 64-character boundary. The QTF editor surmounts these problems and provides some extra features especially useful in the prose-writing process.

So tune in next issue, and in the meantime let's look at some of the benefits of the QTF approach.

Formatting Extensibly

To illustrate how nice it is to have FORTH underneath you, Figure 2 shows the formatting source that produced the glossary in Figure 1. (Actually Figure 2 is a reproduction of the display produced by the QTF Editor.)

Let's look at the source in Figure 2, and see what it does:

The document begins with a comment, just like a regular FORTH screen, useful in recognizing the text when doing an **INDEX**. Naturally, the comment uses FORTH's ordinary parenthesis.

Next we find a FORTH definition! We're defining the word **means** to do a hanging-indent at tab-position 15. We'll see how this definition is used momentarily.

After the definition comes the command **start** which initializes certain variables and moves the printhead down several lines from the top of the page.

Next we find the format command **center[**, following the text to be centered on the page. (The square-bracket is part of the word **center[** to remind us that it also indicates the beginning of some text.)

After some more titles, we format the first word in our glossary, which happens to be the word "start," by delimiting it on both sides by square brackets. Here is where our special purpose word **means** comes in: it tabs over to the second column, the glossary description.

Using this type of special-purpose format command has definite advantages. Not only does it make the source easy to read ("this-command" **means** "that-action"), but it also lets us fiddle with the tabbing distance. Even after we've completed the entire document, we can shift the description column around by changing only one number, in the definition of **means**.

My experience in using this approach has taught me to use many such special-purpose formatting

Continued

Figure 3

```
Screen # 33          crc ver = 53368
 0 ( Quick Text Formatter                                    08/09/82 )
 1 : FORMFEED   12 EMIT ;  ( <-- write code for your own printer)
 2 78 CONSTANT PAPER  ( 80-column width)
 3 ( left, right, top and bottom margins:)
 4 10 CONSTANT LMARGIN          PAPER 10 - CONSTANT RMARGIN
 5  6 CONSTANT TMARGIN          55 CONSTANT BMARGIN
 6 VARIABLE DELIMITER ( current delimiter character)
 7 VARIABLE XTRA  ( amount to indent on auto cr's)
 8 VARIABLE ACROSS  ( horizontal position; absolute)
 9 VARIABLE DOWNWARD  ( vertical position; absolute)
10 : SKIP ( n)  DUP  ACROSS +!  SPACES ;
11 : \line  ( begin new line at appropriate left margin)
12   0 ACROSS !  CR  1 DOWNWARD +!  LMARGIN XTRA @ + SKIP ;
13 -->
14
15


Screen # 34          crc ver = 31098
 0 ( Quick Text Formatter                                    08/09/82 )
 1 : start   ( begin page at top margin; use at start of document)
 2   0 DOWNWARD !  TMARGIN 0 DO \line LOOP ;
 3 : newpage ( begin next page)   CR FORMFEED  start ;
 4 : cr  ( begin next line; if at bottom, start new page)
 5   DOWNWARD @  BMARGIN > IF  newpage  ELSE  \line  THEN ;
 6 : crs ( # of crs -- )  0 DO  cr  LOOP ;
 7 : pp  ( start new paragraph)  cr cr ;
 8 : tab ( n)   ( skip to position "n", relative to left margin)
 9   ACROSS @  LMARGIN -  - 1 MAX  SKIP ;
10 : indent ( n)  ( tab and reset left margin, until next delim.)
11   DUP XTRA !  tab ;
12 : hang-ind ( n)  ( indent; subsequent lines indent 3 more)
13   DUP tab  3 + XTRA ! ;
14 -->
15


Screen # 35          crc ver = 13904
 0 ( Text Formatter                                          08/09/82 )
 1 : ?NEAR  ( -- true: near right)  ACROSS @  RMARGIN > ;
 2 : ?WRAP  ( -- true: at very edge)  ACROSS @  PAPER = ;
 3 : LETTER  ( -- current char.)  BLK @ BLOCK  >IN @ +  C@ ;
 4 : FLUSH-LEFT  ( after a cr, don't output a 2nd blank)
 5   cr  LETTER  BL = IF  1 >IN +!  THEN ;
 6 : PARSE  ( c)    ( display text to delimiter "c" within margins)
 7   DELIMITER !  BEGIN  LETTER  1 >IN +!  DUP
 8   DELIMITER @ =  >IN @ 1023 = OR 0=  WHILE  DUP EMIT
 9   1 ACROSS +!  BL = ?WRAP OR  IF  ?NEAR  IF  FLUSH-LEFT
10   THEN  THEN  REPEAT DROP  0 XTRA ! ;
11 : [  ASCII ]  PARSE ;
12 : (  ASCII )  PARSE ;
13 -->
14
15


Screen # 36          crc ver = 54848
 0 ( Text Formatter Extensions                               08/09/82 )
 1 : sub   cr  5 hang-ind ;
 2 : subsub   cr  10 hang-ind ;
 3 : center[   ( center between margins)  >IN @  RMARGIN LMARGIN -
 4   5 +  ASCII ] WORD C@ -  2/ tab  >IN !  [ ;
 5 : r[  ( n)  ( right justify in field "n" wide)  >IN @  SWAP
 6   ASCII ] WORD C@  -  0 MAX  SKIP  >IN !  [ ;
 7 : load   LOAD ;
 8
 9
10
11
12
13
14
15


Copyright 1982 Leo Brodie                                    09/01/82
```

words for each document. The idea is to describe the format of a document in logical terms, rather than procedural terms. For instance, if you begin each title of a chapter subsection with the word

subsection[

or some such descriptive name, then at any time you can change the definition of **subsection[** to print the section title in boldface, or perhaps to underline, or perhaps to begin the section on a new page.

(When you do define special words, there's a peculiar caution that I almost hate to bring up: some FORTH systems print a message such as "is redefined" or "isn't unique" when a word is defined twice. [My personal opinion is that this test should be user-switchable...] On such systems, you can't print the same document twice without getting the error message in the middle of your document, unless you take some special measure. For instance, you could define all your special formatting words in a separate block, and only load the block once. Or you could define a dummy word called **tabs** on the top of the text formatter application, and begin each document with the phrase

FORGET tabs : tabs ;

or some such.)

Text Management

A few more words about working with the block "modules":

Since printing a document consists of **LOAD**ing a series of FORTH blocks, you can use any method that your system provides for doing that. If you're preparing a short document (one or two pages), it's fine to use a series of blocks, linking one to the next with the word

-->

But for longer documents, such as book chapters or long articles, it turns out to be much easier to provide a "load block." This block contains the main title, and all the section titles, but between section title will be a "load statement" (e.g. **98 load**). By sticking to this approach, you give yourself a directory to all the blocks in your document. It's easy to move sections around, if necessary, by just changing the load order. (The QTF Editor lets you move text strings from block to block, too.)

Within each section, blocks can be

linked together by arrows, or alternatively, you can define the word thru to load a range of blocks from the load block.

Notice that you can format (load) any block at any time. You don't have to start from the beginning of the document. That's the other advantage of using a load block: you can review the entire document by loading the load block, or just sections by loading the section blocks.

Sometimes it's convenient to define the load-blocks to frequently-used documents as **CONSTANT**s. In fact, for my correspondence I have a block reserved for my return address. This block is named as the constant **address**, so when I write a letter, I begin the block with

```
start address load
```

then continue with the date, the addressee's name, etc. Saves me from having to remember where I live.

Suggested Extensions

There are a number of niceties I've left out of this published version. For one thing, I wanted to keep it simple. For another, those niceties are reserved for the product version (which I plan to sell for about $30). But I'll give you some hints — if you want to spend the time you can add them yourself.

Of course, there are all the obvious extensions, such as page numbering, item numbering, automatic heading/footing, justification, etc.

Perhaps less obvious additions are a user-switchable "log" facility, which optionally prints the corresponding block numbers beside the text in the formatted output. This makes later editing extremely easy. Also, an escape from formatting the document during output can be very handy.

And of course there are routines to take advantage of your printer's capabilities. My own verison, for the Epson printer, can print boldface words in the middle of normal text, or lines or even paragraphs in double-width, compressed, or both at the same time, keeping the natural margins of the paper, and of course do underlining.

A word you'll definitely want to add right away is **print**. It should take a block number on the stack, direct your computer's output to the printer, **LOAD** the given block, then return output to your console. This is system-dependent stuff, so I don't show it here.

If you type it in...

I've included checksum verifications at the top of each screen. See the article on checksums elsewhere in this issue.

This application is designed to run on FORTH-79 Standard. To run it on a fig-FORTH system, you should predefine:

```
: VARIABLE  0  VARIABLE ;
: CREATE  VARIABLE  -2 ALLOT ;
: >IN   IN ;
: WORD   WORD  HERE ;
```

Some systems may require that you predefine:

```
32 CONSTANT BL
: ASCII
  BL WORD 1+ C@
  [COMPILE] LITERAL ;
    IMMEDIATE
```

(where **WORD** is the 79-Standard version)

Finally, remember that the Formatter will be frustrating to use until you have the Editor. Can you wait till the next issue?

Notice

Acknowledgements

I thank FORTH, Inc. for permission to use this syntax, which was originally invented by Charles Moore. The entire text of *Starting FORTH* was done on a word processor that used this general approach.

Leo Brodie is an author, lecturer and consultant specializing in FORTH. He teaches the Advanced Systems Course at Inner Access Corporation, and has written system documentation for three FORTH vendors. He is also editor of FORTH Dimensions Magazine. He no longer dances to '50's music but instead listens almost exclusively to Steely Dan records.

Z-80® and 8086 FORTH

PC/FORTH™ for IBM® Personal Computer available now!

FORTH Application Development Systems include interpreter/compiler with virtual memory management, assembler, full screen editor, decompiler, demonstration programs, utilities, and 130 page manual. Standard random access disk files used for screen storage. Extensions provided for access to all operating system functions.

Z-80 FORTH for CP/M® 2.2 or MP/M ..... $ 50.00
8086 FORTH for CP/M-86 ..... $100.00
PC/FORTH for IBM Personal Computer $100.00

Extension Packages for FORTH systems

Software floating point $100.00
Intel 8087 support (PC/FORTH, 8086 FORTH only) $100.00
AMD 9511 support (Z-80, 8086 FORTH only) $100.00
Color graphics (PC/FORTH only) $100.00
Data base management $200.00

Nautilus Cross-Compiler allows you to expand or modify the FORTH nucleus, recompile on a host computer for a different target computer, generate headerless code, and generate ROMable code with initialized variables. Supports forward referencing to any word or label. Produces load map, list of unresolved symbols, and executable image in RAM or disk file. No license fee for applications created with the Cross-Compiler! Prerequisite: one of the application development systems above for your host computer.

Hosts: Z-80 (CP/M 2.2 or MP/M), 8086/88 (CP/M-86), IBM PC (PC/DOS or CP/M-86)
Targets: Z-80, 8080, 8086/88, IBM PC, 6502, LSI-11

Cross-Compiler for one host and one target $300.00
Each additional target $100.00

FORTH Programming Aids by Curry Associates. Includes Translator, Callfinder, Decompiler, and Subroutine Decompiler. 40 page manual. Used with Cross-Compiler to generate minimum size target applications.

Specify host system $150.00

Z-80 Machine Tests Memory, disk, console, and printer tests with all source code in standard Zilog mnemonics ..... $ 50.00

All software distributed on eight inch single density soft sectored diskettes, except PC/FORTH on 5¼ inch soft sectored single sided double density diskettes. Micropolis and North Star disk formats available at $10.00 additional charge.

Prices include shipping by UPS or first class mail within USA and Canada. Overseas orders add US$10.00 per package for air mail. California residents add appropriate sales tax. Purchase orders accepted at our discretion. No credit card orders.

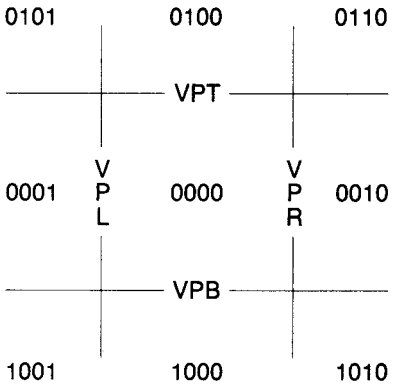
Laboratory Microsystems
4147 Beethoven Street
Los Angeles, CA 90066
(213) 306-7412

The Sheer Joy of Clipping Recursively

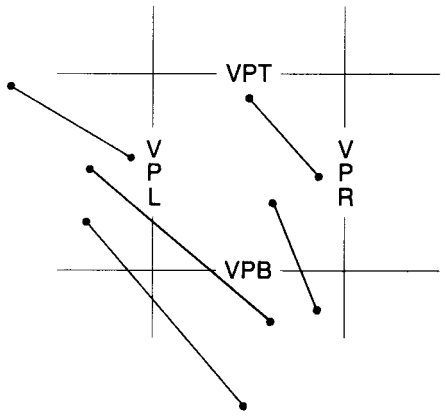
Bob Gotsch

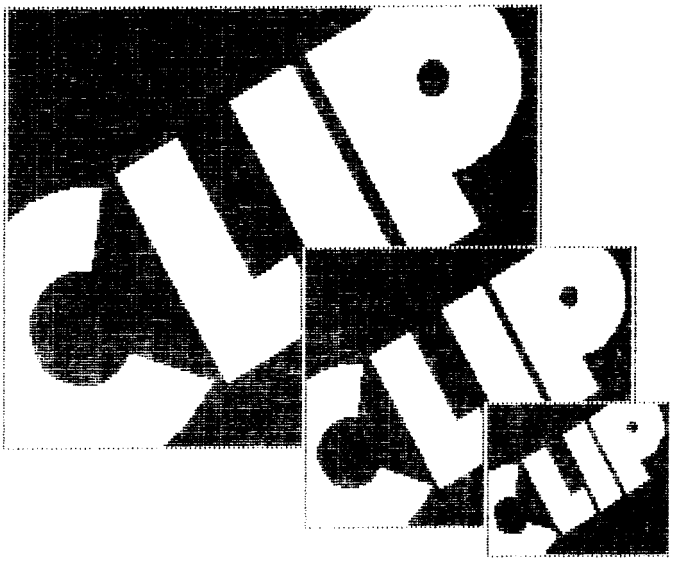
I've been wondering since I wrote the last article whether recursion would be useful for anything else but a study of recursion. That other thing turned out to be "clipping," discarding those parts of the lines of a picture that lie outside the specified "viewport," so that plotting takes place only within the desired or useable portion of the display device. One strategy for clipping is to perform a binary search for the visible extremes of each line, then plot the segment between.

Using the Sutherland-Cohen algorithm, 4-digit "outcodes" are assigned to the endpoints of a line according to where they lie, inside or outside the viewport.



Inside is 0000. For an X-value to the left of (less than) the viewport boundary, the rightmost bit is set (0001). Below the viewport the leftmost bit is set (1001). The outcode for lower left is the logical OR of left and below (1001), etc. A line can be trivially rejected (no plotting at all) if it lies entirely to one side (outside) of the viewport. An efficient test for rejection, returning a true flag, is the logical AND of the endpoint's outcodes. A line may be trivially accepted and plotted as-is when outcodes for both ends are zero.





Of all the possible locations of a line, entirely inside viewport, entirely outside, one end in, or crossing viewport, every situation is handled by trivial rejection, trivial acceptance, or successive middle divisions of the line until each of the segments of the line can be trivially accepted or trivially rejected. The binary search for the intersection with viewport boundary terminates when the segment becomes so short that the midpoint in integer screen coordinates coincides with one or the other endpoint.

I have chosen to save the three values for each endpoint, X-value, Y-value, and outcode together on the stack, with the out-of-viewport point always topmost on the stack. Hence a true **INVIEWPORT?** condition is followed by **3SWAP**. The other tests **TRIVIALACCEPT?**, **TRIVIALREJECT?**, and **COINCIDE?** as well as calculation of **MIDPOINT**, assignment of **OUTCODE**, and the graphics action **PLOTLINE** do just what they say and should be understandable from the foregoing without listed definitions.

These are incorporated in the recursive procedure **REJECT?** which uses the last outcode as true or false flag. If true, it returns to toplevel **CLIPLINE** having dropped one of the points, or at any lower level having droped off the half of the current line that is entirely outside of the viewport. Between **MYSELF**s is a test that drops the midpoint if it happened to be inside the viewport, so the search can continue onward for the other visible extreme of the line, conducted by the other **MYSELF**. If the actions of a recursive procedure are planned in advance, then the procedure-as-a-whole can be written to follow those rules, and each recursive call can be trusted to follow those rules. If **REJECT?** returns false, then **CLIPLINE** must plot the segment represented by the two endpoints left on the stack.

Continued

```
: RETURN R> DROP  ;
: REJECT? ( 2x,2y,2oc,1x,1y,1oc
                                    --- 2x,2y,TF)
                          ( --- 2xcl,2ycl,2oc,1xcl,1ycl,FF)
   TRIVIALREJECT? IF 3DROP RETURN THEN
   TRIVIALACCEPT? IF RETURN THEN
     INVIEWPORT?
     IF 3SWAP                          (swap endpoints)
       MIDPOINT
         COINCIDE?
         IF >R >R 3DROP R> R> 0 RETURN
         THEN
       OUTCODE 3SWAP
           MYSELF
         DUP NOT IF 3SWAP 3DROP THEN
           MYSELF
       3SWAP                           (swap ends back)
     ELSE
       MIDPOINT
         COINCIDE?
         IF >R >R 3DROP R> R> 0 RETURN
         THEN
       OUTCODE 3SWAP
           MYSELF
         DUP NOT IF 3SWAP 3DROP THEN
           MYSELF
     THEN ;
: CLIPLINE ( 1x,1y,2x,2y --- )
    OUTCODE 5 ROLL 5 ROLL OUTCODE
      REJECT? DUP
      IF DROP 2DROP
      ELSE STRIPOUTCODES PLOTLINE
      THEN ;
```

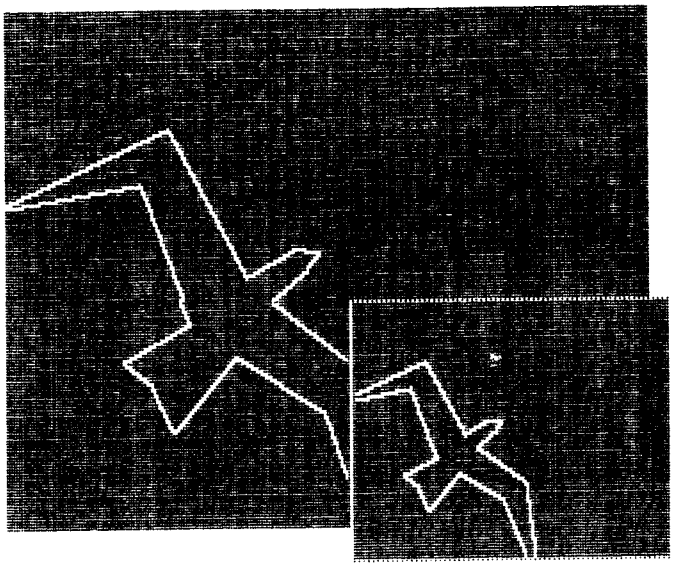
I find it unnecessary (and often almost impossible) to step thru all the levels in planning a recursive procedure; if one reads the toplevel procedure as a "user" of the action-of-the-whole at each of its **MYSELF**s, or reads **MYSELF** as the typical interface between two levels, that should be enough for understanding. But how many levels deep might it actually go in the search for the viewport boundary? To determine the risk of return stack overflow I simulated a high-resolution (1024 wide) graphics display and clipped random lines from a large user space 30000 pixel units wide. Interestingly, and reassuringly, the depth of recursive calls NEVER exceeded 17 — staying well within both parameter stack and return stack limits. Of course a high-level recursive search is slow; the real efficiency of this algorithm would be realized in assembler CODE, calculating the midpoints with two additions and two right shifts.

To illustrate uses of clipping, I have included printouts of a decorative title for this article and a frame from an animation sequence in which a bird and its reduced likeness fly into and out of large and small viewports simultaneously on the screen. □

BIBLIOGRAPHY

1. Foley, J.D. and Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Company, 1982.
2. Newman, W.P. and Sproull, R.F., *Principles of Interactive Computer Graphics*, McGraw-Hill, 1979.

Bob Gotsch is a graphics programmer for Time Arts, Inc. and a teacher of graphic arts at the California College of Arts and crafts. He is interested in exploring the use of computers as aids to artists. He uses FORTHWARE FORTH.



FORTH Based File Handling System

(continued from page 9)

The ability of having several channels active allows easy file to file transfer of information, or simultaneous editing of several files. However, since it would then be possible to have several blocks with the same block number yet on different channels, the routines like **BLOCK** need some very minor alterations to prevent confusion. If there is enough interest from readers, I will discuss these changes in a future article. For now, I recommend that **CHAN** be ignored, and that all file I/O be performed on the default channel zero, and that files be opened, used and closed sequentially.

Conclusion

A FORTH that has the file handling capability has many advantages. The one illustrated is simple, requiring only file string, FDB stuffer and three verbs. Error recovery is as simple. Yet it clarifies FORTH program usage by making source code more modular and circumventing much code since there is no need for documentors or auto-indexes. The effort to add such to FORTH is trivial, due to the modest amount of additional code. The gain is easy file generation, be it FORTH source code, formatted text, target compiled FORTH object code or FORTH generated executable code. □

THE FORTH SPECIALISTS
COLORFORTH AND PCFORTH

Quality figFORTH compilers need not be expensive.

COLORFORTH is a version of figFORTH for the TRS-80 Color Computer. It requires a minimum of 16K, but **does not** require Extended Basic. **COLORFORTH** has been customized for the Color Computer with special DUMP and PRINTER functions and a CSAVEM command for those owners without Extended Basic. When you purchase **COLORFORTH**, you receive both cassette and RS/disk versions, and the figEDITOR. This means no added expense when you upgrade your system. Complete: Both cassette and RS/disk versions with extensive manual. **JUST...$49.95.**

PCFORTH is FORTH tailored for the IBM Personal Computer. You receive all the outstanding qualities of standard figFORTH compiler and editor, plus several additional words to customize it for the Personal Computer. **PCFORTH** requires a minimum of 32K and 1 disk drive (DOS). Complete with diskette and manual. **Only...$89.95.**

DEALER & AUTHOR INQUIRIES INVITED

All items are post paid in U.S. . . . . Texas Residents add 5 percent

Armadillo Int'l Software
P.O BOX 7661
AUSTIN, TEXAS 78712

master charge VISA

PHONE (512) 459-7325

-=-=-=-=-

-=-=-=- 8080/Z80 FIG-FORTH for CP/M & CDOS systems -=-=-=-

-=-=-=-=-

$50 saves you keying the FIG FORTH model and many published FIG FORTH screens onto diskette and debugging them. You receive TWO 8 inch diskettes (single sided, single density, soft sectored only). The first disk is readable by Digital Research CP/M or Cromemco CDOS and contains 8080 source I keyed from the published listings of the FORTH INTEREST GROUP (FIG) plus a translated, enhanced version in ZILOG Z80 mnemonics. This disk also contains executable FORTH.COM files for Z80 & 8080 processors.

-=-=-=-=-

The 2nd disk contains FORTH readable screens including a extensive FULL-SCREEN EDITOR plus many items published in FORTH DIMENSIONS, including a FORTH TRACE utility, a model data base handler, an 8080 ASSEMBLER and formatted memory dump and I/O port dump words. The disks are packaged in a ring binder along with a complete listing of the FULL-SCREEN EDITOR and the FIG-FORTH INSTALLATION MANUAL (the language model of FIG-FORTH, a complete glossary, memory map, installation instructions and the FIG line editor listing and instructions).

-=-=-=-=-

This entire work is placed in the public domain in the manner and spirit of the work upon which it is based. Copies may be distributed when proper notices are included.

-=-=-=-=-

		USA	Foreign AIR
☐	Above described package	$50	$60
☐	Printed Z80 Assembly listing w/ xref..... (Zilog mnemonics)	$15	$18
☐	Printed 8080 Assembly listing.....	$15	$18
	TOTAL	$_____	_____

-=-=-=-=-

Price includes postage. No purchase orders without check. Arizona residents add sales tax. Make check or money order in US Funds on US bank, payable to:

Dennis Wilson c/o
Aristotelian Logicians
2631 East Pinchot Avenue
Phoenix, AZ 85016
(602) 956-7678

-=-=-=-=-

FORTH-83 DO-LOOP

Robert L. Smith

A new form of the **DO-LOOP** has been accepted for the next FORTH standard, tentatively called FORTH-83. The new **DO** will generally work as you would expect for indices which represent either addresses or signed or unsigned arithmetic values. The index **I** covers a complete 65K range, the same as in FORTH-79 but twice as much as in fig-FORTH or poly-FORTH. An additional advantage occurs with **+LOOP**: the sign of the increment can change within the loop without necessarily causing an exit condition. The speed of the new form is faster than most previous loops unless **I** occurs frequently. A feature of the new loop is that when **LEAVE** is executed, control is passed to the end of the loop without intervening calculations.

The price to be paid for the general form of the new loop is that certain "side-effects" of the old form are missing. Consider the simple definition:

```
: TEST  0 DO  I .  LOOP ;
```

Under the old form, **–5 TEST** would execute exactly once. In the new form, the loop continues until the index **I** crosses the boundary between limit and limit-1. In the above case, **–5 TEST** would print out:

```
0 1 2 ... 32767 –32768 –32767 ...
–8 –7 –6
```

To print only one value would require **1 TEST**. For another example, consider the following function:

```
: CLEAR  DO 0 I C! LOOP ;
```

Suppose that our base is hexadecimal and we wish to clear memory between 2000 and EFFF. With the new form of **LOOP**, we could simply type:

```
F000 2000 CLEAR
```

and the indicated area would be cleared. The routine would only clear one byte with the FORTH-79 or the fig-FORTH version of **LOOP**.

The new loop considers that the index **I** lies on a "number circle" based on the usual 2's complement arithmetic for 16 bit numbers. Thus there is a smooth transition between –1 and 0 and between 7FFF and 8000 hex (32767 and –32768 decimal).

There are a variety of ways to implement the new loop, some of which remain to be discovered. There are two parts to the problem. One is to find a method of calculating the exit conditions, and the other is to allow **LEAVE** to work properly. The fastest method of determining the exit conditions requires that the actual value of **I** be calculated by an addition or subtraction. The items stored on the return stack (or elsewhere) are related to the limit and the index, but are not necessarily the same. For machines with a testable overflow bit the suggested technique is to modify the limit and initial index so that the transition will lie between 7FFF and 8000 hex. The overflow bit is set whenever an addition causes the result to cross the 7FFF to 8000 boundary. Initially put

```
limit'  <--  limit + 8000
I'  <--  init – limit'
```

To calculate **I**, note that

```
I = I' + limit'
```

At **+LOOP**,

```
I'  <--  I' + increment
```

Then check for overflow. If the overflow bit is set, continue to loop, else exit from the loop.

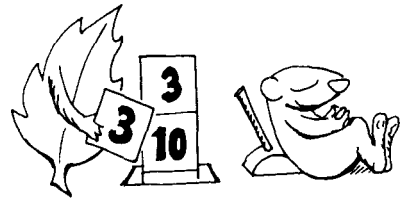
For machines without an overflow bit, such as the 8080, let

```
limit'  <--  limit
I'  <--  init – limit'
```

For **LOOP**, merely increment **I'** by 1 and branch back if the result is non-zero. For **+LOOP**, one has to examine the combination of the carry bit and the sign bit of the increment. Klaus Schleisiek suggests the following: use the RAR instruction to shift the carry bit into the accumulator, then use XRA with the increment value. Only the sign bit of the result is of interest. If the result is positive, continue to loop. If the result is negative, terminate the loop.

There have been various suggestions for implementing **LEAVE**. Bob Berkey's original suggestion involves having the run-time operator for **DO** place the exit address for the loop on the return stack to be used by **LEAVE**. Klaus Schleisiek improved that by having **LEAVE** be an immediate word. By using the return stack at compile time to store the addresses of "fixup" locations, it is possible to avoid run-time penalties when **LEAVE** does not occur in the loop. Bill Ragsdale has suggested a simple but clever way of avoiding use of the return stack (except as a very temporary storage place), since Klaus's method may not be compatible with certain systems. Bill's technique links the forward references in a simple chain and then resolves the chain when the **LOOP** or **+LOOP** is encountered at compile time. As a result of his work, Bill has also suggested an alternative form of **LEAVE**, called **?LEAVE**, which appears to be more useful than **LEAVE** itself. **?LEAVE** takes the top element from the parameter stack and terminates the loop if the element is non-zero (i.e., true). Further details will probably be presented at the next FORML meeting.

One interesting possibility for augmenting the new **DO** is to add a function called, say, **?DO**. When the arguments to **?DO** are equal, as in the case **0 0 ?DO**, then the loop is not executed at all. If that appears sufficiently useful, then that function could be incorporated in **DO** itself, so that an additional word would not be needed. It would require a slight amount of additional time at the beginning of each loop, and would eliminate one (admittedly rarely used) case from **DO**. □



Reprinted from Starting FORTH, *by Leo Brodie, permission of Prentice-Hall, Inc.*

FORTH-79 Compatible LEAVE for FORTH-83 DO . . . LOOPS

Klaxon Suralis
Sunnyvale, California

Editor's note: This paper originally appeared in slightly different form on the Northern California FORTH teleconference tree, July 21-22, 1982.

The major controversy over the proposed FORTH-83 **DO..LOOP** is the implementation of **LEAVE**. Allow me an attempt to explain the problem, and to outline the solution I prefer.

The old **LOOP** and **+LOOP** operate in two separable steps: first, add the increment to the index; then, compare that new index value to the limit. Static and easily understood, but the differing flavors of numeric comparison (signed **<**, unsigned **U<**, circular **– 0<**) beget a multitude of variations: 32K circular **+LOOP**, unsigned-incrementing **/LOOP**, unsigned-decrementing **\LOOP**, and that horror of horrors, the FORTH-79 signed **LOOP** and **+LOOP**.

What's wild is: the same form of **LEAVE** works for all of 'em. FORTH-79's **LEAVE** works by equating the loop's limit parameter to the current index. Thus **I** continues (pardon the grammar) to return its expected value while execution proceeds normally until a **LOOP** or **+LOOP** is encountered. At that point, loop termination is assured (in nearly all practical cases), whether the increment is positive or negative.

The new **LOOP** and **+LOOP** fuse the addition and the comparison steps into one indivisible calculation. The flag for go/no-go arises as a natural by-product of adding increment to index. If, during any one transition, the index crosses the invisible boundary between limit and limit-1 (in either direction, then the loop terminates. Else, you branch back for another go-round.

What's the payoff for all this confusion? For one thing, the new **LOOP** and **+LOOP** can run much faster than their two-step predecessors. For another, the same looping word will work for signed OR unsigned parameters. It's compatible with all earlier **LOOP**s simultaneously, except in certain rare, unsavory cases.

That quirk: the new **LOOP**s lack any common-sense conception of "less than/greater than." If you, mistakenly or not, reverse the start and limit values, or make them equal, the new loop will promenade the long way around the 64K number circle. Oh, well, that may sometimes be useful.

The overflow status in most CPUs (8080s can fake it) offers an efficient, ready-made mechanism for detecting our boundary crossings — *provided* the boundary is always 32, 767.5. The trick, then, is to make **DO** translate its limit and starting values from "true" external form to an adjusted internal form, positioning the loop limit over the 32K overflow threshold and adding an equal offset to the index. This is done only once when the loop is entered.

Later, when we hit **LOOP** or **+LOOP**, we add the increment to the *internal* index and look for an overflow. No comparisons are necessary, and **LOOP** can use the "increment memory" instruction found on most processors. Fast, simple, and no exceptions.

Of course, **J** and **I** must undo the translation by subtracting the offset. Thus, we still carry two loop parameters on the return stack: the adjusted internal index, which changes; and the translation offset, which remains constant.

Although this looping technique was presented just last November at Asilomar by Robert Berkey, acceptance has been swift. It has been incorporated into the FORTH-83 Draft Standard.

Now, if it weren't for **LEAVE**, the proposed FORTH-83 **DO..LOOP** would be too-good-to-be-true. The old **LEAVE** mechanism — efficient, utterly FORTH-like, and devoid of compile-time trickery — simply cannot work with the dynamic boundary-crossing exit criterion of the new **LOOP** and **+LOOP**.

Here's why: there is *no way* to zap those loop parameters on the return stack to guarantee an overflow for every possible increment **+LOOP** might throw at us. For plain **LOOP**, it's easy; but in the general case, it's impossible.

Two currenlty proposed solutions both require a drastic change in **LEAVE**'s function: It must discard the loop parameters and jump straight-away to the end of the looping structure. It sounds rather nice; **LEAVE** would actually *leave*, instead of fooling **LOOP** / **+LOOP** into wrapping things up. How comfortable and easy to teach! As orthodox as Pascal and apple pie.

Never mind an historical discontinuity impeaching every FORTH program ever written; only a very few occurrences of **LEAVE** will actually require a change in coding. You'll find them as you go along. Concentrate instead on how **LEAVE** will know where to jump. These are the two approaches:

In one, **LEAVE** is **IMMEDIATE.** It compiles **(LEAVE)** and leaves space for a branch address which is filled in by **LOOP** or **+LOOP**, much the way **WHILE** compiles **0BRANCH** and allots a displacement later resolved by **REPEAT**. Of course, we'll want to allow multiple **LEAVE**'s — or none at all — within a given **LOOP** structure. So, **LOOP** and **+LOOP** must be coded to handle this, possibly sharing their backfilling routine with **REPEAT** and **UNTIL**. It's complicated, and it costs you extra memory every time you code **LEAVE** — but at least there's no speed penalty at run-time.

The other technique keeps **LEAVE** non-**IMMEDIATE**, as required by FORTH-83 Working Draft 'A.' The requisite jump address is available as a third parameter on the return (loop) stack — not in-line as above. It is the duty of **DO** — **(DO)** actually — to set up that pointer at the same time it computes the translation offset and inter-

nal index. How does **(DO)** know what forward jump address to push? A single in-line displacement, reserved by **DO** and resolved by **LOOP** or **+LOOP**. There's a little more execution-time overhead here than above, but only within **(DO)**, which does its thing just once at the loop's beginning. Perhaps more significant is the second method's appetite for return stack space: six bytes per nesting level.

Folks could argue for weeks over which of these approaches is better; it's quite an entertaining web of tradeoffs. A Standard specifying an **IMMEDIATE LEAVE** could allow either implementation to comply, imposing a mere few extra dictionary bytes (**: LEAVE COMPILE LEAVE ; IMMEDIATE**) on implementors who prefer the second way.

But this is still profoundly incompatible with the way **LEAVE** has worked in fig-FORTH, polyFORTH, FORTH-79, and others. So I'm fueling the debate by presenting a completely different solution.

Remember the reason for messing up **LEAVE** in the first place; there was no setting of those two 16-bit loop parameters which would deliver the "call it quits" message to **LOOP** and **+LOOP** in all conceivable cases.

We'd have to split **LEAVE** into two words: **+LEAVE**, which would work when the increment was known > **0**; and **–LEAVE**, which would handle all negative increments. In addition to being even less compatible with existing code than the immediate **LEAVE**, it makes life miserable for programmers who enjoy switching increment signs within their **+LOOP**s. Anyway, neither **+LEAVE** nor **–LEAVE** would leave correctly for **0 +LOOP**.

These same disadvantages prevail if we ban the word **LEAVE** and replace it with some kind of CHANGE–LOOP–LIMIT word. Ugh.

MY PROPOSAL, in its general form, is to keep an extra piece of information on the return (or loop) stack: a **LEAVE** flag. This flag would be initialized by **(DO)**, set by **LEAVE**, and tested by **(+LOOP)**

There is one quite elegant way to do this, which I'll present at FORML Asilomar this Fall. A discussion of standards, however, should avoid concentrating on any one person's implementation. The individual implementor of a standard system must have the freedom to use the peculiarities of his/her CPU/environment to best advantage.

The **LEAVE** flag needs only one bit, but may be padded to byte or cell width. For reentrancy's sake, it must reside on the return (or loop) stack, or on some stack of its own. Since there are no naturally disallowed combinations of internal index and translation offset, the flag cannot be encoded into the four bytes already assigned to loop parameters.

Thus, compatibility with old **LEAVE** costs us another byte or two on the return stack for each level of nesting (unless you use ROR, ROL, and TST instructions to implement a dedicated one-bit-wide **LEAVE** flag stack somewhere). On the other hand, dictionary space requirements are eased compared to the jumping **LEAVE**s described earlier. User programs are smaller because no forward-branch displacements are embedded in the code — and the system is smaller 'cause it doesn't need complicated **IMMEDIATE** words to mark and resolve them.

If you count clock cycles, loops using **LEAVE** flags run slower than equivalent ones running with immediate **LEAVE**. While I can't deny that a microsecond saved is a microsecond earned, I'll show the price of compatibility with old **LEAVE** to be minimal.

Here's the real issue: assume you've got a **DO LOOP** (or loops) which runs too slow. How often will converting that loop from a **LEAVE**-flag implementation to an immediate **LEAVE** version give you the improvement you need? Next to never, I'd say. Using a faster CPU, going to DTC, or putting NEXT in-line after each system CODE definition would return more on your investment. Anyway, if your loop is so tight and critical that the time eaten by **(LOOP)** or **(+LOOP)** bothers you, it's probably worth translating into native machine code.

(From this point of view, the new **DO..LOOP** is more valuable for its dual signed/unsigned capability than for its questionable performance increase.)

Let's compare **LEAVE**-flag loops versus immediate **LEAVE** loops in more detail:

(DO) executes only once to initialize the loop. The added overhead of pushing a **LEAVE** flag can be ignored.

LEAVE is rarely used, almost never more than once within a given loop. If we complicate it, nobody will notice.

(LOOP) stands to suffer the most from testing an extra flag. However, if we put some extra functions into **LEAVE**, we can eliminate any need for **(LOOP)** to inspect that flag. In addition to setting the flag, **LEAVE** must set the internal index to 32,767 (65,535 on 8080s), and adjust the translation offset so that **I** remains correct. Do this, and **(LOOP)** can be coded exactly the same as in immediate-**LEAVE** implementations.

I and **J** will run equally fast in both systems. A **LEAVE** flag, if present, is irrelevant except perhaps in determining return stack offsets for **J**.

It is **(+LOOP)** which bears the main impact of the **LEAVE** flag, testing it every time. Even here, however, the net effect is practically nil. For example, the overall degradation of the unusually tight loop "**DO I DUP ! 2 +LOOP**" is less than 6%. (This calculation is for my 6809 DTC FORTH, but other CPUs should fall in the same neighborhood.)

I want boundary-crossing loops. For my personal taste, the complexity and incompatibility of a direct, jumping **LEAVE** are intolerable. Although flagged **LEAVE** can never match the old **LEAVE** for simplicity and speed, it is something I can live with.

The existing **LEAVE** usage can work with the new **DO..LOOP**. We don't have to adopt a direct-jumping **LEAVE** in order to reap the benefits of boundary-crossing loop termination. If the immediate **LEAVE** is to become standard, let it do so strictly on the basis of its own merits. I sez it LEAVES much to be desired.

But which **LEAVE** do you prefer? Make up your mind, 'cause there's not enough room in the standard for both compatible and jumping versions.

Hey: Does the Pascal/FORTRAN/COBOL/Ada crowd ever have arguments over this kind of stuff?

CHAIRMAN OF THE BOARDS

INTRODUCING P-FORTH

The P-FORTH Card is the key member in a family of control systems cards offered by the innovators at Peopleware Systems, Inc. P-FORTH has four major advantages:

1. It is a versatile building block. The simple addition of a power supply and terminal makes the P-FORTH card both a low cost development system as well as a target system.
2. An integrated high-level interactive language allows for fast software development.
3. Application programs are stored automatically in non-volatile memory.
4. The STD BUS interface allows the use of a variety of existing peripheral cards.

PUTTING P-FORTH TO WORK:

Users interactively develop applications through a combination of hardware and software. These applications are automatically programmed into non-volatile memory (EEROM). When the applications are proven and functioning, a single switch transforms the system from the developmental mode into the target system.

PEOPLEWARE SYSTEMS INC.
5190 West 76th St.
Mpls., MN 55435 USA
(612)831-0827 • TWX 910-576-1735

SOFTWARE

- An interactive high-level language following the fig-FORTH model
- A monitor for system checkout
- "FORTH-type" screen editor for developing application programs
- A "FORTH-type" assembler for writing assembly language routines
- High-level interrupt-linkage
- High-level communications protocol for down loading from a host system.

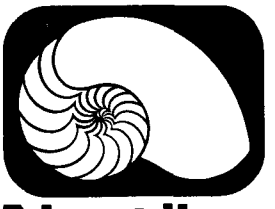
HARDWARE

- 6801 microprocessor
- 6K EEROM
- 8K FORTH firmware
- 2K RAM
- STD BUS interface
- RS232 serial I/O
- 16 TTL I/O lines
- Programmable timer

CROSS-COMPILE FORTH ON APPLE OR ATARI®

The 6502 version of Nautilus Systems Cross-compiler is now available on Apple and Atari. The 6502 target may also be generated on any other host we support. The Nautilus Cross-compiler has been used on every major 8 and 16 bit micro computer.

APPLE or ATARI version $300.00
(plus $5.00 shipping & tax where applicable)
6502 target for existing users $100.00



IBM PC and CP/M 86 versions available from LABORATORY MICROSYSTEMS.
79-Standard versions are available from MOUNTAIN VIEW PRESS.

Nautilus Systems

P.O. BOX 1098 SANTA CRUZ, CA 95061

Apple is a trademark of Apple Computer, Inc. Atari is a trademark of Atari Computer. CP/M is a trademark of Digital Research. IBM PC is a trademark of IBM, Inc.

Defining Words III

Henry Laxen

This is the third and final chapter in my series on Defining Words. In the first episode, we saw how to use defining words to define a simple "Adventure" like interaction. It illustrated how defining words are used in most real world examples. In the second episode we saw how to create a defining word that can define other defining words. Unfortunately, this was only done for 2 levels. This time we will take a look at the ultimate generality, namely iterated defining words. When you read and understand this example, you can consider yourself a black belt defining words champion. I must admit that I have never actually used a construct like this in a real world application, however I think it is a very good exercise in mental gymnastics and the effort put into understanding this will be repaid a hundred times over in the many simpler examples that you will encounter in your FORTH career.

When you read and understand this example, you can consider yourself a black belt defining words champion.

The problem I pose is the following, implement a Tree structure with defining words such that each node in the tree is a word and when it is executed it defines its children. The original defining word will define the root of the tree. You will have to reread that 3 times at least. Before we descend to FORTH code, let's look at some pictures. In Fig. 1 we see what we ordinarily think of as a tree (inorganic). We have a root with 3 children. The leftmost child has 2 children of its own, while the rightmost child of the root has 4 children. This is the kind of tree picture you see in many textbooks, including Knuth, etc. However do not be lulled into thinking that this is the only way trees can be drawn. Let me point out some of its disadvantages. It seems that each parent may have an arbitrary number of children. Unless you have some kind of slick dynamic storage allocation handy, this can be a real pain. Furthermore, with this kind of structure it is very difficult to answer a question like, Who is my sibling? (I don't want to be sexist and say sister.)

Fig. 1

Regular Tree Structure

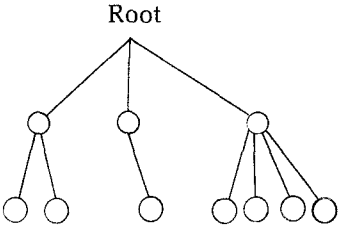


Figure 2 shows the same tree, but with a different representation. Instead of each node having an arbitrary number of points, namely to each of its children, we make each node contain exactly two pointers. One to its next sibling, and one to its first child. Note that as in Fig. 1, the leftmost child of the root has 2 children of its own, and the rightmost child of the root has 4 children. This structure also has some disadvantages. We can no longer answer the question: Who is the nth son of a node, as quickly as we could with the structure in Fig. 1, for now we must search through a linked list to find the answer. However now we no longer need a dynamic storage allocator, and we can quickly find the next sibling of a particular node. Life is full of tradeoffs. We will use the representation described in Fig. 2 for our solution.

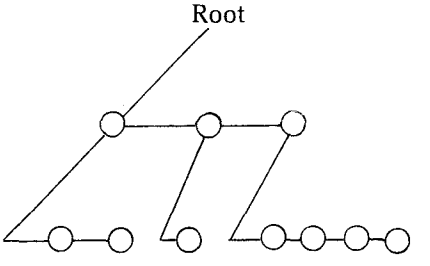
Now let's try to figure out in English what this beast is supposed to do. Starting at the top, the thing that defines the root should set up the root word so that it has no siblings and no children. Put another way, we want to create a defining word, called **ROOT**, which initializes 2 pointers to empty. That much is trivial. The FORTH code for it is simply:

```
: ROOT  CREATE  0 , (Siblings)
  0 , (Children)
```

Thus we have specified the compile time behavior of the word **ROOT**. Now we want to take a look at the run time behavior of the word defined by **ROOT**. Suppose we said **ROOT COLOR**. What is it we want **COLOR** to do if we said **COLOR RED**? Well, we want **COLOR** to **CREATE** the word **RED** and then to link it somehow into the child field of color. Furthermore, we want to initialize **RED** to have no children of its own. Before we do this, let's take a look at Fig. 3 which describes the action of **LINK**. It is given an address and it inserts a new link into a linked list after that address. Convince yourself that the code in Fig. 3 performs the task depicted. This is the same **LINK** word we used in Defining Words II — the **CLASS** example. Now then, armed with **LINK**, let's continue our definition of **ROOT** by specifying the run time behavior of its member word, namely **COLOR**.

Fig. 2

Different Tree Structure



Note the trees in Fig. 1 and Fig. 2 are equivalent.

```
: ROOT
   CREATE  0 , (Siblings)
     0 , (Children)
   DOES>
     CREATE (name of child)
     2+ LINK
       (into Children field of Parent)
     0 ,
       (Initialize my own Children field)
```

Fig. 3

```
: LINK  ( addr -- )
   HERE  OVER @ ,  SWAP ! ;
```

Action of **LINK**

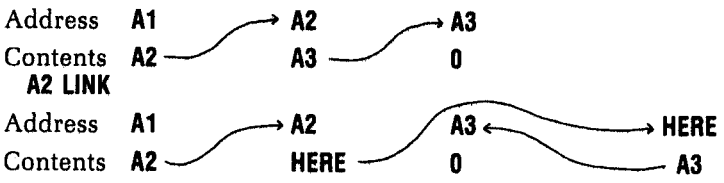


Fig. 4

```
Scr # 27
  0 \ Defining word for Trees                              25JUL82HHL
  1 : LINK    (S addr -- )
  2    HERE    OVER @ ,    SWAP !    ;
  3 : ROOT
  4    CREATE    0 , ( Siblings ) 0 , ( Children )
  5    BEGIN
  6       DOES>     CREATE    2+ LINK ( Siblings )    0 , ( Children )
  7    AGAIN    ;
  8
  9
 10
 11
 12
 13
 14
 15
```

Fig. 5

```
Scr # 28
  0 \ Display Tree in INORDER Sequence                     25JUL82HHL
  1 : MYSELF    (S -- )
  2    LATEST PFA CFA ,      ; IMMEDIATE
  3 VARIABLE INDENT
  4 : .NAME    (S PFA -- )
  5    CR INDENT @ SPACES     NFA ID.    ;
  6 : INORDER    (S PFA -- )
  7    BEGIN    @ DUP WHILE    DUP .NAME
  8       3 INDENT +!    DUP 2+ MYSELF    -3 INDENT +!
  9    REPEAT    DROP    ;
 10 : .TREE    (S -- )
 11    CR 0 INDENT !    ' DUP .NAME CR    2+ INORDER    ;
 12
 13
 14
 15
```

Fig. 6

```
Scr # 29
  0 \ Sample Tree definition                               25JUL82HHL
  1 ROOT POET
  2 POET KEATS    POET SHELLEY    POET BYRON    POET MILTON
  3 KEATS ENDYMION    KEATS NIGHTINGALE    KEATS HYPERION
  4 SHELLEY ALASTOR    SHELLEY ADONAIS    SHELLEY PROMETHEUS
  5 BYRON CHILDE-HAROLD    BYRON MANFRED    BYRON DON-JUAN
  6 MILTON PARADISE-LOST    MILTON LYCIDAS    MILTON ALLEGRO
  7 ENDYMION GOOD    NIGHTINGALE GREAT HYPERION OKAY
  8 ALASTOR GOOD ADONAIS GREAT PROMETHEUS GOOD
  9 CHILDE-HAROLD OKAY MANFRED GOOD DON-JUAN GREAT
 10 PARADISE-LOST OUTSTANDING LYCIDAS OUTSTANDING ALLEGRO GREAT
 11 ( Everyone's a critic )
 12
 13
 14
 15
OK
```

Let's take a close look at this. After **ROOT COLOR**, we have defined the word **COLOR** and set the contents of its PFA and PFA+2 to 0. The contents of the PFA points to the next sibling, and the contents of the PFA+2 point to the first child. Now when **COLOR RED** is executed, the first thing that happens is that **DOES>** supplies us the address of the PFA of **COLOR** on the parameter stack. That's what **DOES>** does, remember? Next we execute the word **CREATE**, which will scan the input stream and make a dictionary entry for the word **RED**. We still have the PFA of **COLOR** on the stack. The **2+** increments it so that now it is pointing at the Children field in **COLOR**, i.e., PFA+2. We link our own PFA into this field. Why is it our PFA that we are linking? Well, **CREATE** makes a dictionary entry for us, up to and including the code field, but it did not do anything to the parameter field. Thus when **LINK** is executing, the PFA of **RED** equals **HERE**, which is what **LINK** will link into the list. Thus after the **2+ LINK** executes, the child field of **COLOR** will have a pointer to the PFA of **RED**, and the sibling field of **RED** will have the former contents of the child field of color, namely **0**.

Next, we execute the **0**, which simply initializes the child field of **RED** to null. Now let's take a look at what happens if we said **COLOR WHITE**. Pretty much the same thing actually, except some of the pointers have changed. **CREATE** would make a dictionary entry for **WHITE**, and the **2+ LINK** code would insert **WHITE** into the child field of **COLOR**. But now, the child field of **COLOR** contains a pointer to the sibling field of **RED**, after all that is what **RED** put there. Thus **COLOR** will point to the sibling field (PFA) of **WHITE**, which will point to the sibling field (PFA) of **RED** which points to **0**. We have succeeded in making **RED** and **WHITE** children of **COLOR** and siblings of each other. Notice that the entries in the tree are in reverse order from how they are defined. Tough.

Pretty slick so far eh? There is only one problem, and that is that it only works for one level. We can use **COLOR** to define a bunch of colors, such as

RED WHITE and **BLUE**, but the buck stops there. When **RED** executes, nothing much will happen. (Exercise: What exactly would happen if we executed **RED** with the above definition?)

Well, let's see what we want **RED** to do when executed. Suppose we said **RED ROSE**. We would like to add **ROSE** to the dictionary and link it into the child field of **RED**. Isn't that exactly what we wanted **COLOR** to do? Of course it is, so what we would really like is to have all subsequent words behave exactly the same. What we want is a (perish the thought) **GOTO** before the ; to go back to just before the **DOES>**. Well for all you structured programming fans who thought that FORTH doesn't have a **GOTO** take a look at the screen in Fig 4. It doesn't have one, but it does. There isn't really any looping going on here, I am simply using the **AGAIN** to jump to the **BEGIN** before the **DOES>**. Now all of the many descendants of the word originally defined by **ROOT** will behave exactly the same, and we can build a completely arbitrary **TREE**.

Fig. 7

```
POET

MILTON
   ALLEGRO
      GREAT
   LYCIDAS
      OUTSTANDING
   PARADISE-LOST
      OUTSTANDING
BYRON
   DON-JUAN
      GREAT
   MANFRED
      GOOD
   CHILDE-HAROLD
      OKAY
SHELLEY
   PROMETHEUS
      GOOD
   ADONAIS
      GREAT
   ALASTOR
      GOOD
KEATS
   HYPERION
      OKAY
   NIGHTINGALE
      GREAT
   ENDYMION
      GOOD

OK
```

In Fig. 5 we have some code that will print out the tree in indented form. Fig. 6 contains a sample Tree definition, and finally Fig. 7 shows the result of printing out the Tree defined in Fig. 6 with the code in Fig. 5. I will not explain how and why the code in Fig. 5 works, since in the next issue I will talk about Recursion in FORTH, and it is a pre-requisite for understanding how **INORDER** works.

One final note. The code presented will work with either FORTH-79 systems or systems compatible with Starting FORTH. It will not work as written with fig-FORTH. Feel free to enter it and try it out. Dumping the dictionary at various points and chasing down the pointers by hand will be very illuminating. Best of luck until next time, and until then, may the FORTH be with you. □

Henry Laxen is a independant FORTH consultant.

Technotes

INPUT COMMANDS AS TEXT LITERALS

H.E.R. Wijnands
Rijswijk, Holland

Here is a technique for literal compilation to defer commands in the input-stream. Suppose we made a definition that operates on a string in the input stream. For instance, a file system command **FLIST** which should be followed by a file name. The command **FLIST** gets the string, stores it in a buffer and searches in the file directory for a match, etc. The command **FLIST** is compilable but it always expects its operand in the input-stream on execution. Sometimes this is unwanted and we wish to execute on a operand inserted during compilation. Example: **: HELP FLIST Helpfilename ;** Execution of **HELP** is meant to give a listing of a helpfile.

The new word **COMLIT** which I derived from compile literal does the trick. **COMLIT** compiles the sequence of characters as literals until it encounters the quotation mark. At run time it executes this sequence.

The problem above is solved as:

```
: HELP COMLIT FLIST Helpfilename" ;
```

Another example of delayed execution (notice its order):

```
: TEST ." testing "
    COMLIT TEST1 TEST2" CR ;
: TEST1 ." one " ;
: TEST2 ." and two" ;
```

Although **TEST1** and **TEST2** are not yet in the vocabulary, **TEST** will be compiled. Executing **TEST** after compilation of **TEST1** and **TEST2** gives:

```
TEST
testing one and two.
OK
```

(Source of **COMLIT** is shown in Figure 1.)

To leave arguments on the stack for further processing after the delayed execution, only the return stack is used by **COMLIT**.

Operation of **COMLIT** is quite simple. During compilation it uses ." to store the string in the parameter **(.")** which normally prints the string at run time. This is not our aim since we want to execute it. For this reason **(COMLIT)** is compiled before **(.")**. At run time **(COMLIT)** jumps over **(.")** diverts the input-stream to the string, interprets its contents and finally restores the input-stream vector.

Purists who dislike the uselessness of **(.")** in the object, can employ the routine part on screen 1 which stores the string. Care should be taken because some FORTH implementations increment the return stack before returning, some after returning from subroutine.

COMLIT is perhaps the counterpart of **DIRECT**. **DIRECT** directly executes the string following it, as if it were a definition. This allows direct use of non-direct words like the control structures **IF**, **DO**, etc.
Example:

```
DIRECT 20 10 DO I LIST LOOP ;
```

The idea of **DIRECT** is not mine, but the source on below 102 is.

This looks useful. I would suggest looking at how ." is defined and creating a similar word, rather than wasting two bytes in each definition. The name **COMLIT** *strikes me as inadequately melifluous. Perhaps something like* **COMPILE"** *would be better.* **DIRECT** *is very similar to Bill Ragsdale's word ::, except that* **DIRECT** *compiles the temporary code in* **TIB**, *while :: compiles it at* **HERE**.

—Michael Perry

EXPONENTIATION

George Lyons
Jersey City, New Jersey

Below (Figure 2) is a possible definition for interger exponentiation, using repeated squaring of the argument, to reduce the number of multiplications.

This does indeed run faster than its simpler **DO . . . LOOP** *equivalent. Although the 16-bit result will overflow for all but the smallest exponents, single-length math lets the heart of the technique show through. Thus its extension to higher precision or floating point is readily apparent. —Klaxon Suralis*

Figure 1

```
 0 ( COMPLETE SOURCE OF DIRECT AND COMLIT )
 1
 2 FORTH DEFINITIONS   HEX
 3
 4 : (COMLIT)   R> IN @ >R >R  R 3 +  TIB @ - DUP >R IN !
 5    INTERPRET  IN @ R>  - 5 + R>  + R> IN !  >R ;
 6
 7 : COMLIT
 8    ?COMP  COMPILE (COMLIT)  ' ." CFA EXECUTE  0 , ;  IMMEDIATE
 9
10 : DIRECT   ?EXEC  DP @  TIB @ 40 + DP !  !CSP  ]  INTERPRET
11    TIB @ 40 + >R  DP !  SMUDGE ;
12
13
14
15 ;S
```

Figure 2

```
 0 : **  ( x n -- x-to-the-n )
 1    >R 1 SWAP  ( square repeatedly )
 2    BEGIN  ( cumulative prod.,last factor )
 3     R 1 AND  ( I. th bit of  n )
 4     IF  ( apply and preserve factor )
 5       SWAP OVER *  SWAP THEN
 6     R> 2/  ( next bit of n ) -DUP
 7     WHILE >R DUP *  ( new factor )
 8    REPEAT DROP ;
 9
10  ( 2/ must be machine code shift )
11  ( extendable to higher precision
12    and floating point x's )
13
14
15
```

NEW PRODUCT ANNOUNCEMENTS

ATAFORTH

The FORTH computer language designed for the Atari home computers. ATAFORTH was written by Dan Bloomquist.

The source text for the system is included. Everything from the compiler core up can be stripped off and be recompiled in less than one minute. This will allow you to modify or dedicate the system at will. Then the new system can be saved as the boot system.

There is a files utility that allows you to "take" sectors from the Atari Disk File Manager, which then sees them as if in use. You can then read and write data directly to these sectors. The directory is bypassed. The utility draws a sector map in graphics, using different colors to show the status of each sector.

Our user's manual is tutorial with numerous examples and is designed for the beginner. All the standard FORTH words have been used so there is no relearning. You work your way from adding numbers on the screen to maintaining a mailing list and producing graphics.

Minimum system configuration: Atari 800/400 with 16K and disk.

$75.00 for disk and manual. (California residents add sales tax.) If interested in a cassette version, contact Dan Bloomquist.

Nova Technology • P.O. Box 688
Clearlake, CA 95422
(707) 994-4649 • 994-1332

SEATTLE COMPUTER PRODUCTS 8086 fig-FORTH

8086 fig-FORTH modified to interface FORTH to SCP's 86-DOS and also compatible with Microsoft MS-DOS and IBM PC-DOS, is available to anyone interested in learning more about FORTH. This version of FORTH is useable (I am currently developing some applications using it), but is not as complete nor as well-documented as a commercial product.

Additional changes to the source align definitions so that word pointers can always be accessed in one bus cycle, add code level array and string primitives, and support fetch and store operations outside the memory segment occupied by FORTH.

Typical execution times on the IBM personal computer are comparable to 6 MHz Z-80 benchmarks which have been published. On the SCP system at 8 MHz, execution times are one-half to one-quarter the Z-80 times. The Sieve of Eratosthenes benchmark (see FD 3:181) runs in just under 34 seconds.

Includes the assembly source, the assembled system, and some FORTH screens (including the FIG editor and the Starting FORTH editor) on disks for any of the above systems or in CPM format, for $35.

Joe Smith
Univ. of Penn./Dept. of Chemistry
34th & Spruce Streets
Philadelphia, PA 19104
(215) 243-4797

COURSE REVIEW

Inner Access Corporation Advanced FORTH Systems Class 6/14-6/18/82

Reviewed by John Clark
San Jose, California

This is a great class! It will be very easy to recommend this class to anyone requiring a detailed knowledge of the internals of FORTH or of the sevral uses of FORTH covered in this class. The small class size was great.

Leo Brodie is very easy to talk with and was always able to understand what we were having trouble with. He also was able to jump right in and dig out how FIG FORTH worked when we found out that FIG didn't work like the system he was experienced with. His use of the computer to illustrate the operations going on inside of FORTH while words were being compiled was very easy to follow and made it much easier to understand FORTH.

If there were more time, I'd like to see more on style, compiler security, multitasking, and performance monioring tools. But if this course had everything I wanted, it would take more than a week.

John Clark works for an international computer firm with offices in San Jose, California.

Laxen & Harris, Inc. Terminate Operations

Effective immediately, Laxen & Harris, Inc. is ceasing operations and no more classes or services will be offered directly by Laxen & Harris, Inc. Our instructional disk set (working FORTH and Learning FORTH) will continue to be available under license through Mountain View Press, P.O. Box 4656, Mt. View, CA 94040; (415) 961-4103.

Both Henry Laxen and Kim Harris will remain active in the FORTH community and will be offering consulting services. If you would like to contact them individually you will find them listed under Consultants in the Vendor's List of *FORTH Dimensions* magazine.

THE FOURTH SOURCE™

NEW FORTH PRODUCTS

☐ **Personal FORTH** for the IBM-PC by FORTH Inc. Multitasking, full screen editor, floating point support, DOS file handler, color monitor support, turnkey compiler. $300

☐ **MULTI-TASKING FORTH** CP/M, Northstar & Micropolis. A-FORTH by Shaw Labs, Ltd. can operate your micro like a mainframe. Print, sort, and inter-activly input, all at the same time $395

☐ **FORTH TUTORIAL** by Laxen & Harris. Two 8" CP/M disks with documentation and a copy of "Starting FORTH" by Brodie. The easy way to learn FORTH. $95

☐ **"And so FORTH"** by Huang. An indepth how-to book about FORTH with a Z80 implementation. Follows the fig-FORTH model. $25

MORE FORTH DISKS

FORTH with editor, assembler, and manual. •Source provided. Specify disk size!

☐ **APPLE II/II +** by MicroMotion $100
☐ **APPLE II** by Kuntze• $90
☐ **ATARI®** by PNS $90
☐ **CP/M®** by MicroMotion $100
☐ **CROMEMCO®** by Inner Access $100
☐ **HP-85** by Lange• $90
☐ **IBM-PC®** by Laboratory Microsystems $100
☐ **PET®** by FSS $90
☐ **TRS-80/I®** by Nautilus Systems• $90
☐ **6800** by Talbot Microsystems $100
☐ **6809** by Talbot Microsystems $100
☐ **Z80** by Laboratory Microsystems $50
☐ **8086/88** by Laboratory Microsystems $100

Enhanced FORTH with: F-Floating Point, G-Graphics, T-Tutorial, S-Stand Alone, M-Math Chip Support, X-Other Extras, 79-FORTH-79. Specify Disk Size!

☐ **APPLE II/II +** MicroMotion, F, G, & 79 $140
☐ **CP/M** by MicroMotion, F & 79 $140
☐ **H89/Z89** by Haydon, T & S $250
☐ **H89/Z89** by Haydon, T $175
☐ **PET** by FSS, F & X $150
☐ **TRS-80/I or III** by Miller Microcomputer Services, F, X, & 79 $130
☐ **6809** by Talbot Microsystems, T & X $250
☐ **Z80** by Laboratory Microsystems, F & M Each $100
☐ **8086/88** by Laboratory Microsystems, F & M Each $100

CROSS COMPILERS Allow extending, modifying and compiling for speed and memory savings, can also produce ROMable code. •Requires FORTH disk.

☐ CP/M $200
☐ H89/Z89 $200
☐ TRS-80/I $200
☐ Northstar® $200
☐ IBM• $300
☐ 8086• $300
☐ Z80• $300
☐ 6809 $350

☐ **fig-FORTH Programming Aids** for decompiling, callfinding, and translating. $150

fig-FORTH Model and Source, with printed Installation Manual and Source Listing.

☐ APPLE II® ,5¼
☐ 8080/Z80® , 8
☐ 8086/88, 8
☐ H89/Z89, 5¼
Each $65

MVP-FORTH – A Public Domain Product

MVP-FORTH contains a kernal for transportability, the FORTH-79 Standard Required Word Set, the vocabulary for the instruction book, STARTING FORTH, by Brodie, editor, assembler, many useful routines, and utilities.

MVP-FORTH PRODUCTS for CP/M® IBM-PC® and Apple®

☐ **MVP-FORTH Programmer's Kit** including disk with documentation, ALL ABOUT FORTH, and STARTING FORTH. Assembly source listing versions. $100
☐ **MVP-FORTH Disk** with documentation. Assembly source listing version. $75
☐ **MVP-FORTH Cross Compiler** with MVP-FORTH source in FORTH. $300
☐ **MVP-FORTH Programming Aids** for decompiling, callfinding, and translating. $150
☐ **MVP-FORTH Assembly Source** Printed listing. $20
☐ **ALL ABOUT FORTH** by Haydon. $20

●●★MVP-FORTH operates under a variety of CPU's, computers, and operating systems. Specify your computer and operating system.★★★

FORTH MANUALS, GUIDES & DOCUMENTS

☐ **FORTH Encyclopedia** by Derick & Baker. A complete programmer's manual to fig-FORTH with FORTH-79 references. Flow charted, 2nd Ed. $25
☐ **1980 FORML Proc.** $25
☐ **1981 FORML Proc.** 2 Vol. $40
☐ **1981 Rochester Univ. Proc.** $25
☐ **Using FORTH** $25
☐ **A FORTH Primer** $25
☐ **Threaded Interpretive Languages** $20
☐ **AIM FORTH User's Manual** $12
☐ **APPLE User's Manual** MicroMotion $20
☐ **TRS-80 User's Manual,** MMSFORTH $19
☐ **Starting FORTH** by Brodie. Best instructional manual available. (soft cover) $16
☐ **Starting FORTH** (hard cover) $20
☐ **METAFORTH** by Cassady. Cross compiler with 8080 code $30
☐ **Systems Guide to fig-FORTH** $25
☐ **Caltech FORTH Manual** $12
☐ **Invitation to FORTH** $20
☐ **PDP-11 FORTH User's Manual** $20
☐ **CP/M User's Manual,** MicroMotion $20
☐ **FORTH-79 Standard** $15
☐ **FORTH-79 Standard Conversion** $10
☐ **Tiny Pascal in fig-FORTH** $10

☐ **Installation Manual for fig-FORTH,** contains FORTH model, glossary, memory map and instructions $15

Source Listings of fig-FORTH, for specific CPU's and computers. The Installation Manual is required for implementation. Each $15

☐ 1802 ☐ 6502 ☐ 6800 ☐ AlphaMicro
☐ 8080 ☐ 8086/88 ☐ 9900 ☐ APPLE II
☐ PACE ☐ 6809 ☐ NOVA ☐ PDP-11/LSI-11

Ordering Information: Check, Money Order (payable to MOUNTAIN VIEW PRESS, INC.), VISA, MasterCard or COD's accepted. No billing or unpaid PO's. California residents add sales tax. Shipping costs in US included in price. Foreign orders, pay in US funds on US bank, include for handling and shipping by Air: $5 for each item under $25, $10 for each item between $25 and $99 and $20 for each item over $100. Minimum order $10. All prices and products subject to change or withdrawal without notice. Single system and/or single user license agreement required on some products.
DEALER & AUTHOR INQUIRIES INVITED

MOUNTAIN VIEW PRESS, INC.

PO BOX 4656 **MOUNTAIN VIEW, CA 94040** **(415) 961-4103**

Start Your Own FIG Chapter

What is a FIG Chapter

There are two kinds of FIG chapters: local, and special-interest. Local chapters are centered in a city or region. special-interest chapters may be non-geographical; they focus on an interest area such as an application (e.g., robotics, telecommunication), or on FORTH for a particular computer.

All chapters must provide a contact point, and some form of regular public access (usually meetings). Non-geographical chapters will normally provide other forms of access, such as a newsletter or telecommunications, instead of meetings.

Why Have a FIG Chapter?

A chapter lets you share information with other FORTH users in your geographical or application area. In addition, FIG provides several specific benefits:

(A) FIG will list your chapter in *FORTH Dimensions*, so that others can find your group.

(B) *FORTH Dimensions* will give priority to publishing chapter news, which can help you make professional contacts in the areas of your particular interests.

(C) FIG will occasionally supply material, such as meeting handouts or tapes, which can serve as a discussion topic at local meetings.

(D) FIG will supply its publications at bulk rates; local chapters can sell them to raise money, and to provide immedite local access to the material.

(E) Chapters can apply to FIG for one-time funding for activities.

How to Start a FIG Chapter

To be recognized as a chapter, a group must have (1) a contact person, (2) regular public access (usually by meetings which are open to the public), and (3) at least five members of FIG. If you don't know five members in your area, FIG can help you contact them. If you want to start a chapter, send a request for a FIG Chapter Kit to the Chapter coordinator, FORTH Interest Group, P.O. Box 1105, San Carlos, CA 94070.

FIG Chapter News

Potomac Chapter

At the August 3rd meeting, Joel Shprentz described a keyboard and display controller written in FORTH. Multifield commands are entered on a thirteen-key keyboard (10 numeric and 3 control keys). As each field is entered it is displayed on a 4-digit LED display.

The 24 possible commands require different combinations of fields. Most languages would force the programmer to write either voluminous code or an elaborate table driven system. With FORTH the code is simple and direct; words controlling individual field input are combined in various ways to define higher level words which control groups of fields and entire commands.

Other Chapters?

Let's hear from you!

List of FORTH System Vendors

(e.g., A1 signifies AB Computers, etc.)

Processors

1802	C1, C2, F3, F6, L3
6502 (AIM, KIM, SYM)	R1, R2, S1
6800	F3, F5, K1, L3, M6, T1
6809	F3, F5, L3, M6, T1
68000	C4, E1
8080/85	A5, C1, C2, F4, I5, L1, L3, M3, M6, R1
Z80/89	A3, A5, C2, F4, I3, K1, L1, M2, M3, M5, N1
Z8000	I3
8086/88	F2, F3, L1, L3, M6
9900	E2, L3

Operating Systems

CP/M	A3, C2, F3, I3, L3, M1, M2, M6

Computers

Alpha Micro	P3, S3
Apple	A4, F4, I2, I4, J1, L4, M2, M6, O2, O3
Atari	M6, P2, Q1
Cromemco	A5, M6
DEC PDP/LSI-11	C2, F3, K1, L2, S3
Heath-89	M6
Hewlett-Packard 85	
IBM PC	C2, F3, L1, M5, M6
IBM Other	L3
Micropolis	A2, M2, S2
North Star	I5, M2, P1, S7
Ohio Scientific	A6, B1, C3, O1, S6, T2
Osborne	
Pet SWTPC	A1, A6, B1, C3, O1, S6, T2, T5
TRS-80 I, II, III	I5, M5, M6, S4, S5
TRS-80 Color	A3, F5, M4, T1

Other Products/Services

Boards, Machine	F3, M3, R2
Consultation	C2, C4, N1
Cross Compilers	C2, F3, I3, M6, N1
Products, Various	C2, F3, I5, S8
Training	F3, I3

FORTH Vendors

The following vendors offer FORTH systems, applications, or consultation. FIG makes no judgement on any product, and takes no responsibility for the accuracy of this list. We encourage readers to keep us informed on availability of the products and services listed. Vendors may send additions and corrections to the Editor, and must include a copy of sales literature or advertising.

FORTH Systems

A

1. AB Computers
252 Bethlehem Pike
Colmar, PA 18915
215/822-7727

2. Acropolis
17453 Via Valencia
San Lorenzo, CA 94580
415/276-6050

3. Advanced Technology Corp.
P.O. Box 726
Clinton, TN 37716

4. Applied Analytics Inc.
8910 Brookridge Drive, #300
Upper Marlboro, MD 20870

5. Aristotelian Logicians
2631 East Pinchot Avenue
Phoenix, AZ 85016

6. Aurora Software Associates
P.O. Box 99553
Cleveland, OH 44199

B

1. Blue Sky Products
729 E. Willow
Signal Hill, CA 90806

C

1. CMOSOFT
P.O. Box 44037
Sylmar, CA 91342

2. COMSOL, Ltd.
Treway House
Hanworth Lane
Chertsey, Surrey KT16 9LA
England

3. Consumer Computers
8907 La Mesa Boulevard
La Mesa, CA 92041
714/698-8088

4. Creative Solutions, Inc.
4801 Randolph Road
Rockville, MD 20852

D

1. Datentec Kukulies
Heinrichsallee 35
Aachen, 5100
West Germany

E

1. Emperical Research Group
P.O. Box 1176
Milton, WA 98354
206/631-4855

2. Engineering Logic
1252 13th Avenue
Sacramento, CA 95822

F

1. Fantasia Systems, Inc.
1059 Alameda De Las Pulgas
Belmont, CA 94002
415/593-5700

2. Fillmore Systems
5227 Highland Road
Minnetonka, MN 55343

3. FORTH, Inc.
2309 Pacific Coast Highway
Hermosa Beach, CA 90254
213/372-8493

4. FORTHWare
639 Crossridge Terrace
Orinda, CA 94563

5. Frank Hogg Laboratory, Inc.
130 Midtown Plaza
Syracuse, NY 13210
315/474-7856

6. FSS
P.O. Box 8403
Austin, TX 78712
512/477-2207

I

1. IDPC Company
P.O. Box 11594
Philadelphia, PA 19116
215/676-3235

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

3. Inner Access
517K Marine View
Belmont, CA 94002
415/591-8295

4. Insoft
10175 S.W. Barbur Blvd., #202B
Portland, OR 97219
503/244-4181

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

J

1. JPS Microsystems, Inc.
361 Steelcase Road, West, Unit 1
Markham, Ontario,
Canada L3R 3V8
416/475-2383

L

1. Laboratory Microsystems
4147 Beethoven Street
Los Angeles, CA 90066
213/306-7412

2. Laboratory Software
Systems, Inc.
3634 Mandeville Canyon Road
Los Angeles, CA 90049
213/472-6995

3. Lynx
3301 Ocean Park, #301
Santa Monica, CA 90405
213/450-2466

4. Lyons, George
280 Henderson Street
Jersey City, NJ 07302
201/451-2905

M

1. M & B Design
820 Sweetbay Drive
Sunnyvale, CA 94086

2. MicroMotion
12077 Wilshire Boulevard, #506
Los Angeles, CA 90025
213/821-4340

3. Microsystems, Inc.
2500 E. Foothill Boulevard, #102
Pasadena, CA 91107
213/577-1417

4. Micro Works, The
P.O. Box 1110
Del Mar, CA 92014
714/942-2400

5. Miller Microcomputer Services
61 Lake Shore Road
Natick, MA 01760
617/653-6136

6. Mountain View Press
P.O. Box 4656
Mountain View, CA 94040
415/961-4103

N

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

O

1. OSI Software & Hardware
3336 Avondale Court
Windsor, Ontario
Canada N9E 1X6
519/969-2500

2. Offete Enterprises
1306 S "B" Street
San Mateo, CA 94402

3. On-Going Ideas
RD #1, Box 810
Starksboro, VT 05487
802/453-4442

P

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

2. Pink Noise Studios
P.O. Box 785
Crockett, CA 94525
415/787-1534

3. Professional Management
Services
724 Arastradero Road, #109
Palo Alto, CA 94306
408/252-2218

Q

1. Quality Software
6660 Reseda Boulevard, #105
Reseda, CA 91335

R

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

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

S

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

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

3. Sierra Computer Co.
617 Mark NE
Albuquerque, NM 87123

4. Sirius Systems
7528 Oak Ridge Highway
Knoxville, TN 37921
615/693-6583

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

6. Software Federation
44 University Drive
Arlington Heights, IL 60004
312/259-1355

7. Software Works, The
1032 Elwell Court, #210
Palo Alto, CA 94303
415/960-1800

8. Supersoft Associates
P.O. Box 1628
Champaign, IL 61820
217/359-2112

T

1. Talbot Microsystems
1927 Curtis Avenue
Redondo Beach, CA 90278

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

3. Timin Engineering Co.
6044 Erlanger Street
San Diego, CA 92122
714/455-9008

4. Transportable Software, Inc.
P.O. Box 1049
Hightstown, NJ 08520
609/448-4175

Z

1. Zimmer, Tom
292 Falcato Drive
Milpitas, CA 95035

Boards & Machines Only

see System Vendor Chart for others

Controlex Corp.
16005 Sherman Way
Van Nuys, CA 91406
213/780-8877

Datricon
7911 NE 33rd Drive, #200
Portland, OR 97211
503/284-8277

Golden River Corp.
7315 Reddfield Court
Falls Church, CA 22043

Peopleware Systems Inc.
5190 West 76th Street
Minneapolis, MN 55435
612/831-0872

Zendex Corp.
6398 Dougherty Road
Dublin, CA 94566

Application Packages Only

see System Vendor Chart for others

R.E. Curry & Associates
P.O. Box 11428
Palo Alto, CA 94306

Decision Resources Corp.
28203 Ridgefern Court
Rancho Palo Verde, CA 90274
213/377-3533

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

Consultation & Training Only

see System Vendor Chart for others

Boulton, Dave
581 Oakridge Drive
Redwood City, CA 94062

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

Girton, George
1753 Franklin
Santa Monica, CA 90404
213/829-1074

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

Harris, Kim R.
Forthright Enterprises
P.O. Box 50911
Palo Alto, CA 94303
415/858-0933

Laxen, Henry H.
1259 Cornell Avenue
Berkeley, CA 94706
415/525-8582

Petri, Martin B.
15508 Lull Street
Van Nuys, CA 91406
213/908-0160

Redding Co.
P.O. Box 498
Georgetown, CT 06829
203/938-9381

Dr. Walter Schrenk
Postfach 904
7500 Krlsruhe-41
W. Germany

Software Engineering
317 W. 39th Terrace
Kansas City, MO 64111
816/531-5950

Technology Management, Inc.
1520 S. Lyon
Santa Ana, CA 92705

FORTH INTEREST GROUP MAIL ORDER

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

NAME_____MAIL STOP/APT_____

ORGANIZATION_____(if company address)

ADDRESS_____

CITY_____STATE_____ZIP_____COUNTRY_____

VISA #_____MASTERCARD #_____

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

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

ORDER PHONE NUMBER: (415) 962-8653

FORTH INTEREST GROUP **PO BOX 1105** **SAN CARLOS, CA 94070**

FORTH INTEREST GROUP

P.O. Box 1105

San Carlos, CA 94070

BULK RATE
U.S. POSTAGE
PAID
Permit No. 261
Mt. View, CA

477 940865ITH@R@@@@P
R L SMITH
ESL SUBSIDIAR OF TRW
PO BOX 510
SUNNYVALE, CA 94086

Address Correction Requested