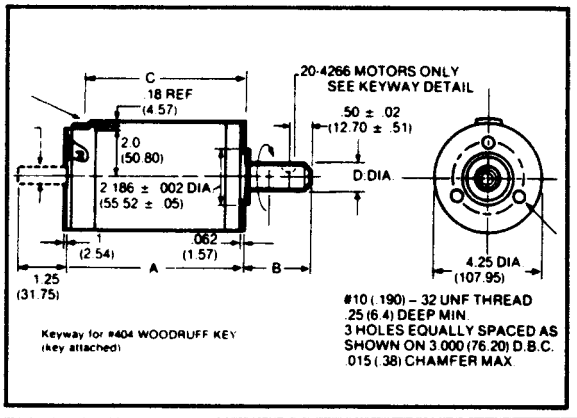
FORTH DIMENSIONS

VOLUME IV, NUMBER 2 $2.50

INSIDE:



HARDWARE CONTROL

Z-80® and 8086 FORTH

FORTH Application Development Systems including interpreter-compiler with virtual memory management, assembler, full screen editor, line editor, decompiler, demonstration programs, and utilities. Standard random access disk files used for screen storage. Extensions provided for access to all operating system functions. 120 page manual.

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

Floating point extensions for above systems. Specify software floating point, AMD 9511, AMD 9512, or Intel 8087 support.... additional $100.00

Nautilus Cross Compiler systems allow 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. (Prerequisite: one of the application development packages above for your host system)

Z-80 host: 8080 or Z-80 target $200.00
Z-80 host: 8080, Z-80, or 8086 target $300.00
8086 or PC/FORTH host: 8080, Z-80, or 8086 target..... $300.00

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 Z-80 or 8086 FORTH screen file or fig-FORTH style diskette $150.00

Z-80 Machine Tests. Memory, disk, console, and printer tests with all source code. Specify CP/M 2.2 or CP/M 1.4..... $ 50.00

AMD-9511 arithmetic processor S-100 interface board.

Assembled and tested, without AMD 9511 ..... $200.00
Assembled and tested, with AMD 9511..... $350.00

PC/FORTH distributed on 5¼ inch soft sectored double density diskettes. All other software distributed on eight inch soft sectored single density diskettes. North Star and Micropolis formats available at extra 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.

Z-80 is a trademark of Zilog, Inc. IBM is a trademark of International Business Machines Corp. CP/M is a trademark of Digital Research, Inc.

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

Letters . . .

Making It In Japan

Dear FIG,

According to a recent report of the Association of Electronic Industry the popularity of various languages in Japan for application software development is changing as follows:

	Ever experienced or using	Wish to use in future work
Assembler	41.7%	2.0%
BASIC	23.1	6.5
PL/M	18.6	14.1
FORTRAN	10.1	6.5
C	1.5	21.6
COBOL	1.0	—
FORTH	1.0	10.6
PASCAL	0.5	32.2
ADA	—	5.0

The figure for FORTH seems surprising considering the lack of integrate publication about this language in Japan.

Toshio Inoue
Professor of Mineral Processing
University of Tokyo

China News

Dear FIG,

It has been almost six months since I last reported on our FORTH discussion group at Taipei. It is alive and well. We are now meeting at the EE Department in the National Taiwan University every fourth Saturday from 2 to 5 p.m. Participants vary from 20 to 50. An encouraging sign of the strength of this group is that people are bringing programs to be distributed in the meetings.

I am also teaching a course on FORTH to the EE seniors in the EE Department of Chung Yuan Christian University. Its head, Dr. Lo, implemented a ROMmable FORTH on his ZDS system, which was used to develop an intensive care unit for local hospitals. It was a success, I just learned.

At this moment, we have about 10 FORTH programmers at the professional level and about 100 enthusiasts. FORTH literatures have been spread to more than 1,000 people. About 20 FIG-FORTH systems are in regular use. FORTH is still far from being a household name here, but it is known to most micro hobbyists.

Dr. C. H. Ting
Taipai, Taiwan

6809 Gift

Dear FIG,

I am herewith releasing my copyright on the 6809 fig-FORTH source code listing and placing it in the public domain to be distributed by the FORTH INTEREST GROUP. You are hereby authorized to alter the listing to give the standard FIG notation that the listing may be copied provided that due credit always be given the source.

Raymond J. Talbot, Jr.
Talbot Microsystems
Redondo Beach, CA

Hunting Figheads

Dear FIG,

Enclosed is my check for membership renewal. I would like to compliment you on the quality and economy of your work and publications.

I have been contacted by a local head-hunter (employment agency) who mentioned that he got my name and telephone listing from a FIG membership list. I do not know whether it was a local or global list. I feel neutral about such a practice presently, provided the head-hunter does not persist in an obnoxious manner after being told to cease and desist. If other FIG members have had similar experiences, and found them objectionable, some guiding policy on distributing member lists to head-hunters vs. to vendors should be discussed.

Larry Pfeffer
San Diego, California

Here is FIG's policy on the utilization of our mailing list. The list is available for rent, but with two conditions: first, FIG itself will do the actual mailing, the renter never actually gets any list. Second, the material has to be approved as being appropriate. Of course, anyone on the list who does not wish to receive such material can make a written request to be excluded from the rental list — our data base has this capability built in.

As for the incident you described, Roy Martens does not remember renting any list to a head-hunter. It's possible that someone could get a list of local members from the local chapter. This would depend on the local chapter's policy. —Editor

VAX And Ye Shall Receive?

Dear FIG,

We are interested in implementing FORTH on our VAX 11-780 computer. I've noticed that among the FORTH vendors there is no reference to the VAX. There is, of course, the package distributed by John James for the PDP-11. We would be able to run this version in compatibility mode, but a version that runs in native mode would have obvious advantages. I would appreciate your help in locating such a vendor.

James H. Rapp
C-010 Computer Center
UCSD, La Jolla, California

Nice Work, Chuck

Dear FIG,

I was very impressed with the last issue of FORTH Dimensions, especially Michael Perry's article on Charles Moore's BASIC compiler. It took me quite a while to figure it out, but when I did, was I ever impressed. It's one of the slickest pieces of software I've ever seen. When people ask me "What makes FORTH so damn good?" I ask them what languages have they worked with that you can write a BASIC compiler in just 8 screens? Keep up the good work.

Marc Perkel
Springfield, Missouri

Letters continued on next page

FORTH Dimensions

Published by FORTH Interest Group
Volume IV, No. 2 July/August 1982

Editorial/Production
Leo Brodie

Publisher
Roy C. Martens

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

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

Letters *(continued) . . .*

Minor Errors

Dear FIG,

I have a few notes on F.D. III-6. In regard to J.T. Currie, Jr.'s letter, good FORTH code, looking neat, improves readability and brings out structure. Since my own code doesn't always live up to this standard, I best not be too preachy keen. In Henry Laxen's article, the stack diagram for **(ASSIGN)** should read (pfa. . .).

I am completely confused by Marc Perkel's article on control structures. Where are **BRANCH** and **0BRANCH**; what is **C,** doing in **IF**; what is C3; why aren't the control words immediate??

In E.H. Fey's article on a general case statement on page 194, screen 171, line 7, there should be a **1+** between the **DUP** and the **C@**. Of course, the **DEFAULT;** at the bottom of the column before should be **DEFAULT:** .

So much for minor errors. Everyone I talked to said it was the best issue of F.D. yet published. We like lots of code and ideas to look at and think about.

Robert Dudley Ackerman
San Francisco
Apple Code FORTH Users

Thanks for the comments, R.D. Regarding Marc Perkel's article on control structures. Marc was referring to (although he didn't say so) the ASSEMBLER's structures. See John Cassady's 8080 assembler in the same issue, and Marc's article will make a whole lot more sense. (C3 is an 8080 JMP instruction in hex.) Also thanks, R.D., for your article on the recursive decompiler, which appears in this issue. —Editor

Poor Documentation

Dear FIG,

I purchased a fig-FORTH model and machine-readable source code from Mountain View Press late last year. Earlier in the year, I purchased Leo Brodie's "Starting FORTH," on the basis of a recommendation that it was the best introduction to FORTH available.

While I was (am) a novice FORTH programmer, I am not a novice computer programmer/system designer, having been involved in systems software development for both minis and micros during the last 6 years. I am writing to you because, having read Leo's book, I had some serious misconceptions about how fig-FORTH operated. I was only able to clear these up after I spent much time and effort reading both ASM-86 and FORTH sources. If you are able to integrate the information provided below with the installation guide, others who follow the same route I did (i.e., Brodie + fig-FORTH) will have much less trouble with "Starting FORTH" than I.

The most important things:

- The treatment of the disk in fig-FORTH appears to be completely different from that described in chapter 10 of Brodie. This distinction becomes important as soon as you want to copy blocks and maintain the disk, because the techniques Brodie describes don't work. A section which describes how fig-FORTH treats the disk and what the extra bytes in a disk buffer are for (referred to in 5.0 of the installation manual) would clear this up, if added to the installation manual.
- EXECUTE or ' (tick) work differently. In Brodie, the following works:
  : GREET ." Hello, I speak FORTH " ; OK
  ' GREET EXECUTE Hello I speak FORTH OK
  (pg. 216)
  In fig-FORTH, you have to say:
  ' GREET CFA EXECUTE
  to get FORTH to return Hello I speak FORTH. (leave out **CFA**, and FORTH crashes).
- Aside from the above, there are two other areas where naming conventions differ:
  — fig-FORTH **DP** is FORTH '79
  — disk-related words (**BLK, BLOCK, SCR,** etc.) don't work as expected from reading Brodie. (I haven't figured this out yet — but I will, eventually — hopefully).
- In the installation manual, the user variables **IN** and **HLD** are not identified with the 'U' identifying them as such. In general, User variables are not well enough described (When do they get changed? Under what conditions?).
- The variable **CURRENT** is not described (It's a user variable).
- The 'parameters' **C/L** (characters per line, 64)) **B/BUF** (bytes per disk sector) and **B/SCR** (buffers per screen, 8) are not defined anywhere (that I could see). The assembler code defines them, however.

On another subject, I plan to build some FORTH words which allow you access to CP/M-86 files as an alternative to the use of FORTH 'screens' and no directories. If you know of anyone already doing this work (or if you are interested in adding it to your repetoire of products), please let me know.

Derek Vair
Weston, Ontario

The situation you're describing is both real and unfortunate. The FIG model was created in 1978 and generously placed in the public domain by its implementors to spread the popularity of the language. The model was not, however, the only version of FORTH around, and the 79-Standard was later adopted to resolve the many differences between these versions.

"Starting FORTH" was written more in accordance with the 79-Standard than with the FIG model. (The book took some exceptions to the 79-Standard, and many of these exceptions are being incorporated into the 83-Standard.) "Starting FORTH" was generously financed by FORTH, Inc., again with the goal to spread the popularity of the language.

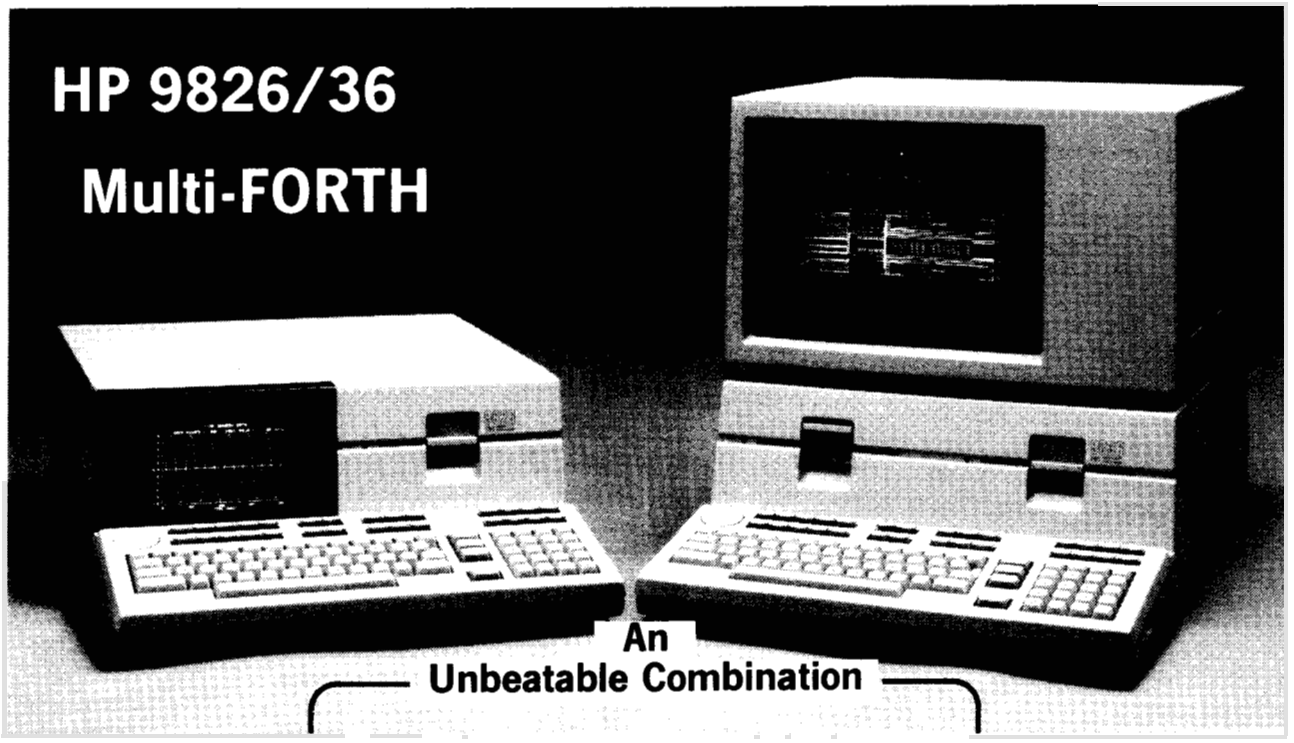
The problem is neither with fig-FORTH, nor its documentation, nor with "Starting FORTH," but simply with the fact that they were created at different times. Many vendors are now selling versions of FORTH compatible with the 79-Standard. Mountain View Press has even placed such an implementation in the public domain, and Glen Haydon has published a book, "All About FORTH" which describes this implementation, carefully noting differences between it and other common versions.

For those who will continue to use the FIG model, perhaps there is some other generous volunteer who will write and publish additional documentation to integrate the discrepancies. —Editor

□

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

FORTH in the Computer Toolbox

Mark Bernstein
Department of Chemistry
Harvard University

```
: WIGGLE
  MOVE 1 MM FORWARD
  MOVE 1 MM BACK
  BEEP ;
```

Listing 1. FORTH lets a user define macro-operations without significant effort. In the example, **WIGGLE** wiggles a translation stage over a precisely controlled distance, an operation useful in testing and lubricating the motor.

A prominent, but misguided, goal of lab computer design has become the "program-free" instrument — one that can plug into an experiment and collect data without explicit programming. To retain some degree of flexibility, a limited "programmability" is achieved through some form of menu selection, using either conventional menu trees or special-purpose, dedicated keyboards to select the actions which may be performed.

In practice, "keystroke function selection" is completely equivalent to programming in a small, very-high-level language. Indeed, in some commercial systems[1] the computer translates each key stroke into a FORTH word for immediate execution. To comfort timid users, the control language is made to be very simple and very rigid.

Despite its superficial appeal, this approach ignores the real needs and abilities of scientific users. Research is unpredictable, indeed often chaotic. Scientists need flexibility, even at the cost of complexity, since the laboratory environment changes constantly as new results demand new methods and techniques. Lab computers ought to promote creative research. Too often, though, rigid pre-programmed instruments actively inhibit creativity, freezing experimental procedures into a fixed, unchangeable mold.

Rather than attempt to provide a "program-free" facility, we have tried to build a "computer toolbox"[2] for our laser spectroscopy work. The toolbox concept embraces hardware and software design, with the goal of providing a powerful and flexible array of tools to knowledgeable, capable users.

We don't avoid programming. On the contrary, we make program writing a commonplace, everyday activity.

We build programs to run experiments, often on the spur of the moment, using existing FORTH and assembly-language procedures. Simple program-building tools, including decent editors and modular device drivers, help make programs easy to write, and easy to document for later reference. Most programs are short, take a few minutes to write and test, and are simple to understand and use, for most laboratory procedures are fundamentally simple. The complexity of lab computing lies not in the complexity or subtlety of individual procedures, but in the vast variety of procedures which may be required.

Toolbox Components

Given adequate tools, it is almost always easier to build a simple homemade program than to subvert a complicated, program-free system which doesn't quite do its appointed task. FORTH, by providing powerful mechanisms for program construction, facilitates impromptu programming and experimenting, and so promotes creativity in the lab.

Most obvious, and perhaps most useful, is FORTH's inherent macro facility. Any important sequence of operations can be given a name and treated as a logical unit (Listing 1)[3]. Temporary macros can be defined at the keyboard, or in "scratch screens"

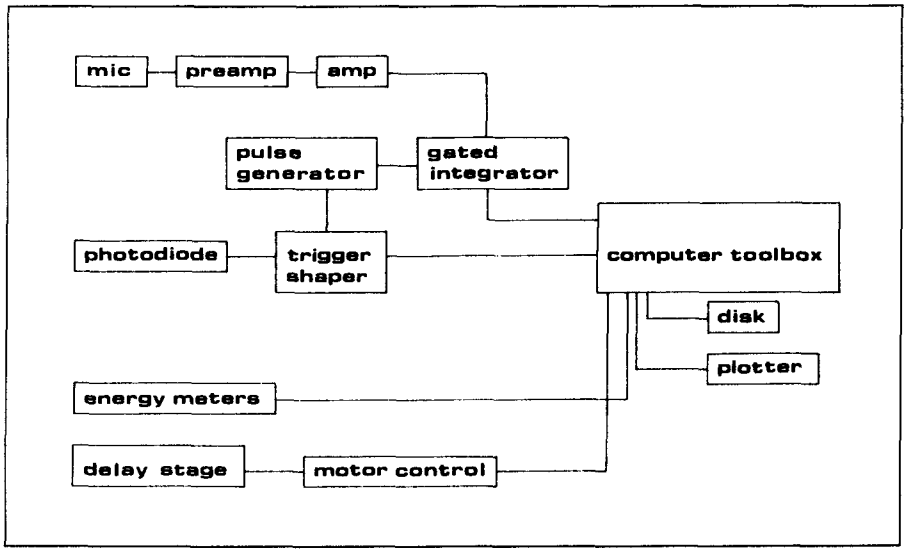


Figure 1. This block diagram represents a typical configuration for our computer toolbox. Note the relative complexity of the computer node — toolbox design encourages the computer to get involved with many aspects of the experiment. The detailed configuration tends to change every few weeks, as instruments and experimental needs change.

Continued on page 7

```
: AVERAGE-POWER ( -> power )
  0
  100 0 DO
    POWER@ +    LOOP
  100 / ;
```

Listing 2. AVERAGE-POWER measures the mean laser power, measured by a device-driver called **POWER@**, by averaging 100 shots. Loops like this take only a few seconds to write, and can provide answers to many unanticipated questions.

— work areas for writing temporary tools. Particularly useful sequences can be stored permanently, to be called up whenever needed. Disk residence is ideal for plotting procedures, data reformatters, and analysis programs that are used intermittently, and are not especially useful in building new tools. FORTH's compactness also allows us to retain a variety of tool components in memory at all times, so that device drivers and tool-building aids are always online and available. In fifteen months of active experimental work, we have yet to run short of dictionary space in our 24K 6502 system.

FORTH's stack architecture provides a convenient mechanism with which to pipeline data from one instrument to another (Figure 2). The stack is also a good place to average data accumulated over many runs; given **POWER@**, a routine which fetches a reading from the laser power meter, we can readily write **AVERAGE-POWER** (Listing 2), which provides the averaged power over 100 laser shots.

It is more pleasing to work *with* the computer rather than against it, to build a procedure out of simpler tools and device drivers than to try to subvert a powerful but rigid procedure which doesn't quite do the job.

Planning For Disaster

The laboratory can be a difficult environment for a computer, especially one which is connected to many independent alien devices. Cables come unstuck or short circuit, power supplies malfunction, ICs fail. Unpredictable, minor failures crop up constantly; the computer must be able to cope gracefully with unexpected mishaps.

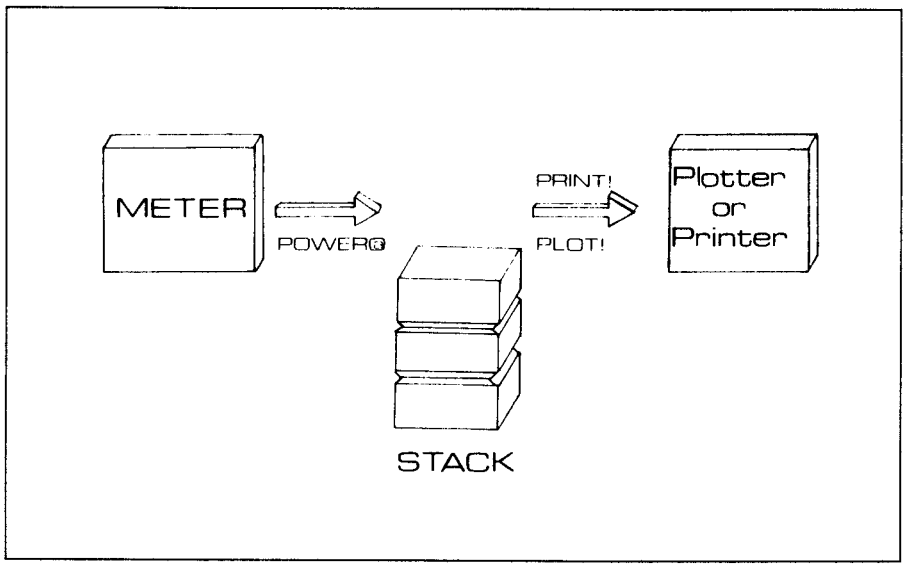


Figure 2. The FORTH stack provides a useful channel for the simplest and most common lab computing function — moving data from one instrument to another. Since **PLOT!** and **PRINT!** are functionally equivalent, they can be interchanged freely as experimental needs alter. The FORTH stack is also useful for format coversion and scaling required by many instruments.

Extra hardware helps accommodate failure or accident. But hardware redundancy is virtually useless unless software design provides consistent support. If the address of parallel port A appears explicitly in dozens of lines of program code, backup ports B and C will not help the unhappy researcher who finds that port A is not working properly. Faced with the task of locating every reference to port A's physical address, users may find it easier to wait for repairs than to use backup equipment which the software cannot easily support.

FORTH's modular structure helps accommodate necessary repairs without demanding extensive program changes. Last winter our laser's grounding system started to fail. A blizzard made repair impossible, and left us with large, fast noise spikes everywhere. Normally, we invoke **?FIRED** to detect laser pulses:

?FIRED (—> f)

?FIRED waits for the laser to fire, and returns a logical flag to indicate whether the laser fired successfully or failed. Laser firing is detected by the photodiode interface bit.

We build programs to run experiments, often on the spur of the moment, using existing FORTH and assembly-language procedures.

Because of the grounding problem, **?FIRED** started to respond to noise spikes as well as to valid laser shots.

Since the offending noise pulses were very short, we thought the computer might be able to adapt to the noise by ignoring anamolously short trigger signals. A 500 microsecond delay was inserted into **?FIRED**, after which FORTH double-checked: was the trigger signal still active? After an hour or so of tinkering, mostly spent fine-tuning the delay time, the computer was able to ignore the noise pulses without missing a single genuine shot.

Continued on next page

Abstract Devices

Most instruments are conceptually simple; they either generate data for the computer to read, or they take instructions from the computer and act upon them. We would like to treat instruments like memory, to be read and written at will. In FORTH terms, we like to think of **PRINT!** and **PLOT!** and **DISK!**, of **VOLTS@**, of **?BUSY** or **?FIRE.**

Of course, the detailed operation of many real devices is complicated by details of interfacing and lack of standardization. Typically, a single experiment may include a dozen different instruments, each demanding unique data formats and transmission protocols. Some instruments accept numbers in binary, some insist on BCD, others require ASCII strings. Data incompatability poses a serious obstacle to communication, since a programmer frequently faces a bewildering array of detail, which she must remember, without fail, at all times.

FORTH encourages designers to use instruments via "device drivers." These procedures translate data from standard FORTH internal representation into the peculiar language understood by individual instruments, mapping the real (complicated) device onto a simple abstract device that behaves the way the user expects. Device drivers hide implementation details, so that users need not remember (or understand) exactly what data format each instrument requires.

In addition to simplifying the programmer's task, device drivers help the system adjust to new and modified equipment. Device-specific information is restricted to the device driver, and not allowed to propagate throughout the toolbox system.

If every program that used the printer handled that printer's peculiarities, then replacing the printer would mean modifying every individual program. Bizarre devices should not be permitted to contaminate and infect system software; their needs should be respected, but their oddities ought to be quarantined within device drivers, where they can be monitored and modified when necessary.

Abstract devices can also provide powerful conceptual aids (Figures 3, 4). For example, our toolbox hardware includes a pair of DACs. What could be more obvious than to connect them to an XY recorder? Of course, we don't want plotting programs to know about the details of the DAC interface; these are hidden in **DAC!**:

DAC! (millivolts chnl —>)

Sets the output of digital-to-analog channel chnl to the indicated voltage.

Much of the time, though, we want to send the plotter to a designated point on the paper. Also, we don't really want to keep talking about "DAC channel 1" when we mean "the X coordinate." So, after a few days we wrote **XY!**:

XY! (x-coord y-coord —>)

moves the plotter pen to a position {x-coord, y-coord}, measured relative to {XORIGIN, YORIGIN}.

Later, we realized that "turtle graphics" were better than cartesian plot routines for some jobs. So the pen became a turtle, whose current heading is stored in a variable **HEADING**, and which responds to commands **FORWARD**, **LEFT**, and **RIGHT**. **FORWARD** is built out of **XY!**, which in turn was built from **DAC!**. Only **DAC!** knows the intimate details of the plotter interface, and users can choose to think of the plotter as either a Cartesian or

Continued on page 11

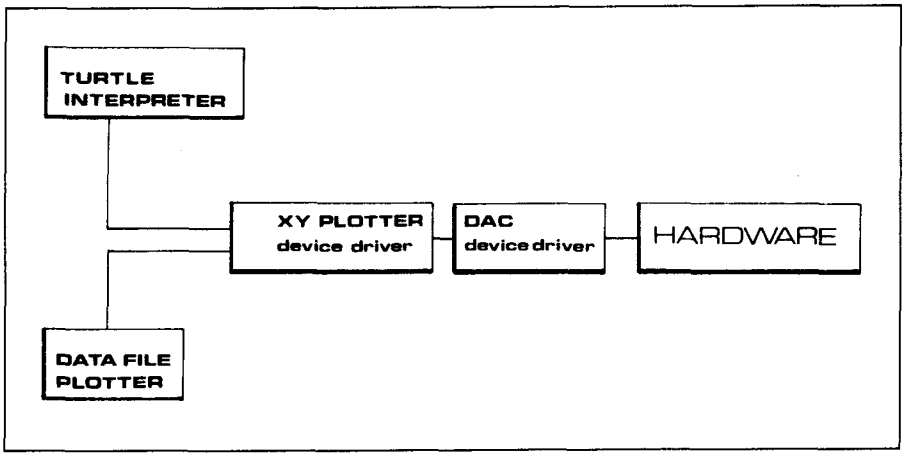


Figure 3. Device drivers promote modular independence by separating hardware from application software. Hardware changes impact only device drivers, not user software. In addition, applications can treat hardware as conceptually simple abstract devices like the turtle, rather than depending on physical hardware characteristics.

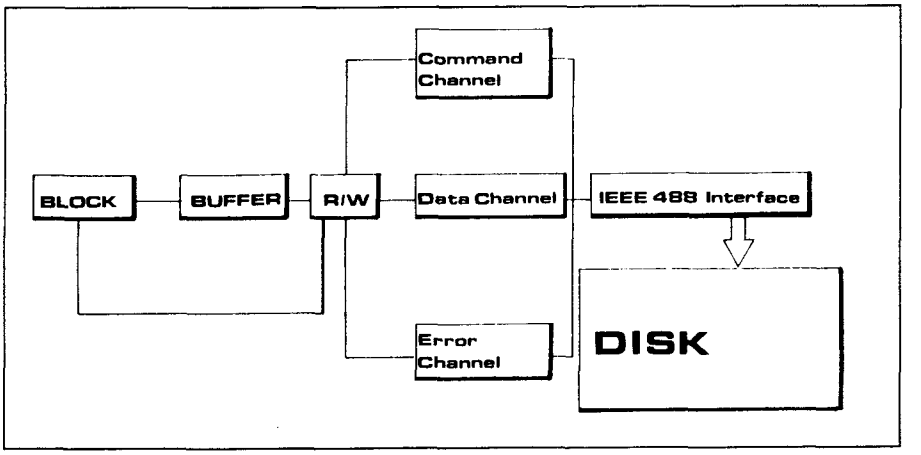


Figure 4. A classic example of an abstract device is the FORTH mass-storage implementation. Programmers almost always work with **BLOCK**, which maps the disk into a conceptually simple format. Within the system, on the other hand, **R/W** may view the disk very differently; here, **R/W** addresses three distinct abstract devices. The abstract devices, in turn, access the disk drive indirectly through device drivers, thus shielding **R/W** and **BLOCK** from future hardward changes.

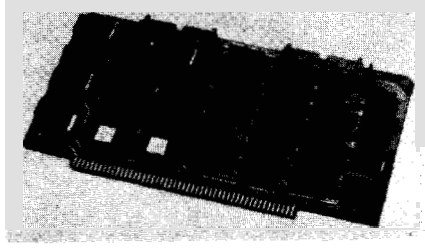
DEVELOPMENT TOOLS

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

VISA

master charge

MVP-FORTH

A Public Domain Product

ORDER TODAY!!!

In keeping with the public domain release of FORTH by its inventor, Charles Moore, and the promotion of the language by the FORTH Interest Group, MVP-FORTH (for Mountain View Press) and the companion book, ALL ABOUT FORTH, are also placed in the public domain and may be used freely without restriction.

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

- ☐ **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. ★ ● ★

MORE FORTH DISKS

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

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

$65 Each

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 +** by Micro-Motion, 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 Micro-systems, T & X $150
- ☐ **Z80** by Laboratory Micro-systems, F & M $150
- ☐ **8086/88** by Laboratory Microsystems, F & M $150

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• $200
- ☐ 6809 $350

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

FORTH MANUALS, GUIDES, & DOCUMENTS

- ☐ **FORTH Encyclopedia** by Derick & Baker. A complete programmer's manual to fig-FORTH with FORTH-79 references. Flow charted $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

THE FORTH SOURCE™

MOUNTAIN VIEW PRESS, INC.

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

A turtle device, without concern for the hardware which underlies these essentially simple concepts.

Command-Driven Systems

Rigid command structures may simplify programming, but ultimately they lead only to frustration. Unless the designer gets everything exactly right, users will need simple features which the machine could obviously manage, but which it cannot be told to do.

Menu and prompt-driven systems are useful, indeed invaluable, in performing and regulating repetitive and well-defined tasks. Both help protect system integrity. But, while a payroll program needs good protection from casual users or systematic abuse, lab computers really ought not to protect themselves from their users, at least not without good reason.

A decent command language is not difficult to learn. Once learned, it should not be difficult to use. FORTH and its kin make command language interpreters very simple to write, since such interpreters simply subset the language. The payoff is substantial; far from being a "write-only" language as is sometimes claimed, in control applications FORTH can be substantially more straightforward than BASIC or FORTRAN.[4]

Implementation

The second generation microprocessors, the 6502, 8080/Z-80 and 6800 machines, are frequently CPU bound in common lab operations. Implementation efficiency is essential, and FORTH's simple lexical analysis, vestigial parser, and threaded architecture[5-7] match the limited computational power of these machines quite well.

In the coming three to five years, newer and more powerful machines should relax the CPU constraints under which lab computers now labor, permitting implementors to overcome some annoying limitations of the fig- and 79-STANDARD systems. By relaxing the CPU's workload, such enhancements will also help simplify laboratory programming, since fewer and fewer tasks will require critical programming to meet their timing restraints.

These developments hold great promise for lab computer design. Greater CPU throughput will permit lab systems to dispense with some FORTH atavisms, notably the strict reverse-Polish grammar necessitated by FORTH's primitive parser and the absence of data typing and run-time security, features now omitted in the interest of increasing CPU throughput. On the other hand, greater CPU power could also be harnessed to the conventional FORTH model, permitting faster data acquisition. But, of greater importance, the next five years should bring a general recognition of the importance of true programmability to creative experimental work, accompanied by further understanding of the role of software tools in small systems.

Acknowlegements

The author expresses his sincere appreciation to Dr. Kevin S. Peters for continued support of this research.

1. For example, the Princeton Applied Research OMA controller, a popular and successful instrument, operates in FORTH. A number of functions have 1-letter names, and so can be invoked by pressing a single key.

2. M. Bernstein, "The Computer Toolbox," *BYTE* (May 1982, p. 456).

3. The motor control commands used in WIGGLE are described in M. Bernstein's "Stepper Motor Control; A FORTH Approach," *MICRO* (February 1982).

4. M. Bernstein and D. P. Gerrity, "Microcomputer Interfacing; FORTH vs. BASIC," *MICRO* (June 1982, pg. 77).

5. Peter M. Kogge, "An Architectural Trail to Threaded-Code Systems," *IEEE Computer* (March 1982, p. 22).

6. James R. Bell, "Threaded Code," *Communications of the ACM 16*, (June 1973, p. 370); J. B. Phillips, M. F. Burke and

7. G. S. Wilson, "Threaded Code for Laboratory Computers," *Software — Practice and Experience 8* (1978, p. 257).

Mark Bernstein is a graduate student in Harvard's Department of Chemistry. His current research combines ultrafast lasers and sensitive microphones, all controlled by a FORTH-based microcomputer, to study fast chemical reactions. □

The FORTH Step
Stepper Motor Control

Application by Martin B. Petri

Text by
Martin B. Petri & Leo Brodie

Stepper motors supply the muscle for computer applications with complicated movements, such as computer disk drives and printers, industrial robots, telescopes, laser systems, etc. Unlike regular motors, stepper motors

Our approach will be to control the stepper motor lines directly. In effect, we will build a driver chip in software.

can be controlled with exacting resolution — on some motors the resolution is .005 degrees of rotation or less. Their speed is also directly controllable, and stepper motors can stop on a dime, or a micrometer.

Operation

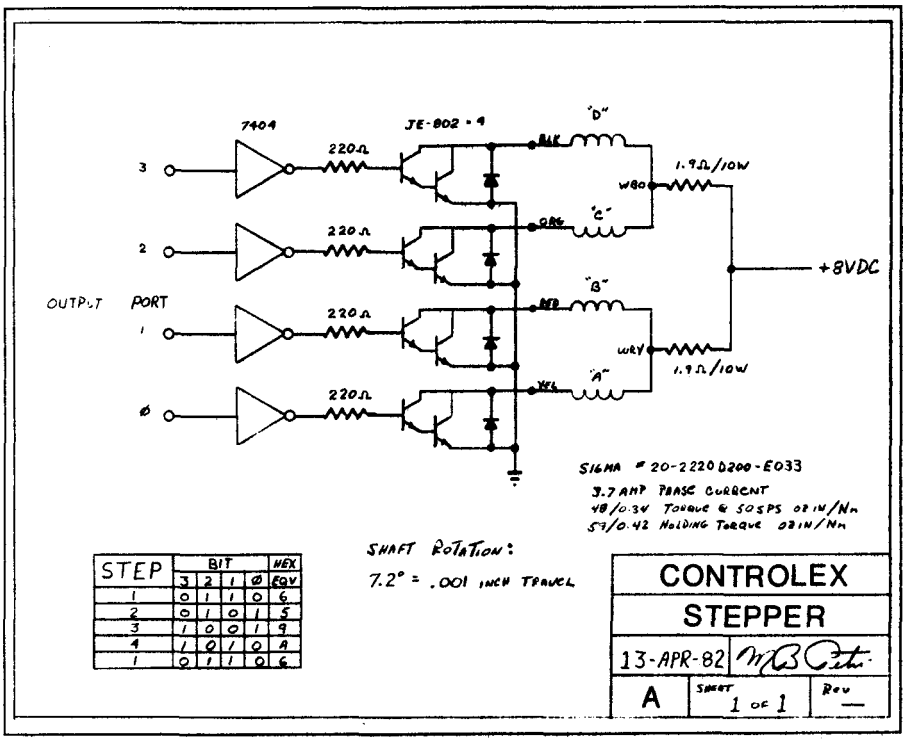
Most stepper motors are controlled by either four or eight lines. In our example, we'll assume a four-pole motor. A "step" is a one-quarter revolution of the internal shaft of the motor — the smallest amount we can cause the motor to turn. Each step corresponds to a certain combination of the motor's four input "poles" being turned on.

Most stepper motors, including the one in our example, contain a gear mechanism. The motor in our example is geared such that one complete cycle produces 7.2 degrees of rotation.

Interfacing Techniques

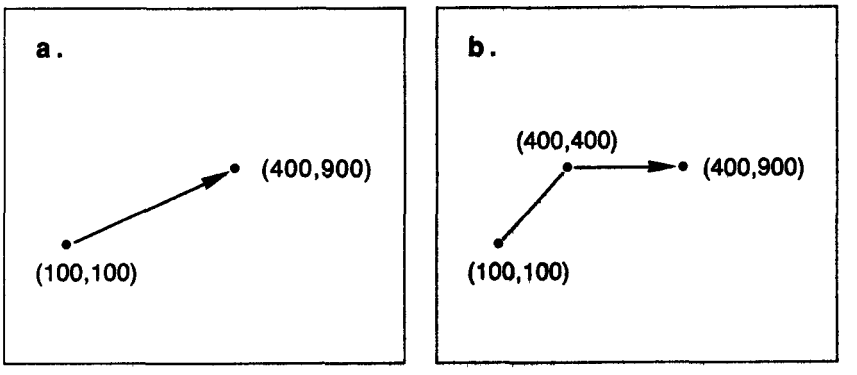
There are several ways to control the stepper motor. One is to buy a stepper motor driver IC. The most advanced of these driver chips allows the computer to simply output the desired motor position. The microprocessor-based chip will then produce the appropriate number of signals on the control lines to move the motor to the stated position.

Figure 1



STEP	BIT 3	2	1	0	HEX EQV
1	0	1	1	0	6
2	0	1	0	1	5
3	1	0	0	1	9
4	1	0	1	0	A
1	0	1	1	0	6

Figure 2



Producing a direct diagonal (Fig. 2a) between the two points above requires the Y-motor to run a sub-optimal speed, and greater complexity of the software. Producing a 45-degree diagonal, then a straight line (Fig. 2b), runs no slower and allows software simplicity. In the latter case, both motors run at full speed in step with one another, until the correct y-coordinate is reached; then the Y-motor shuts off.

```
Screen # 1
  0 ( The FORTH Step      1 of 10    02JUN82MBP )
  1         4 CONSTANT PORT-A         ( address of motor I/O port )
  2        10 CONSTANT MIN-TIME       ( minimum Q-Step time )
  3        10 CONSTANT ACC-TIME       ( acceleration interval time )
  4        15 CONSTANT ACC-STEP       ( acceleration steps )
  5        ACC-TIME ACC-STEP * MIN-TIME + ( calculation for )
  6           CONSTANT MAX-TIME       ( maximum Q-Step time )
  7         0 VARIABLE STEP-TIME      ( holds current Q-Step time )
  8         0 VARIABLE X-POSITION     ( current X position )
  9         0 VARIABLE Y-POSITION     ( current Y position )
 10         0 VARIABLE X-PTR          ( x-phase table pointer )
 11         0 VARIABLE Y-PTR          ( y-phase table pointer )
 12         0 VARIABLE X-COUNT        ( # of steps to go )
 13         0 VARIABLE Y-COUNT        ( # of steps to go )
 14         0 VARIABLE X-DIR          ( direction flag )
 15         0 VARIABLE Y-DIR          ( direction flag )        ;S


Screen # 2
  0 ( The FORTH Step      2 of 10    02JUN82MBP )
  1
  2 : ACCELERATOR             ( --- )
  3         STEP-TIME @       ( get old value )
  4         ACC-TIME -        ( subtract acceleration time )
  5         MIN-TIME MAX      ( but MIN-TIME is minimal value )
  6         STEP-TIME ! ;     ( save new value )
  7 ( Accelerate the Q-Step time until MIN-TIME is reached )
  8
  9 : XY-MOTORS               ( x-addr y-addr --- )
 10         C@ SWAP C@ +      ( xy-phase --- )
 11         PORT-A P!         ( set the new phases )
 12         STEP-TIME @       ( get time value )
 13         0 DO LOOP ;       ( and wait it out )
 14 ( Move stepper 1 Q-step and wait )
 15 ;S


Screen # 3
  0 ( The FORTH Step      SUBSTITUTE for XY-MOTORS )
  1
  2 ( Substitute this screen for XY-MOTORS to get a visual )
  3 ( display of the motor bits and time delay )
  4         ( y-bit     x-bit     timer )
  5
  6 : BINARY 2 BASE ! ;
  7 : (XY-MOTORS)             ( x-addr, y-addr --- )
  8          C@ 16 / CR       ( x-addr, y-phase --- )
  9          BINARY 4 .R      ( display in binary )
 10          4 SPACES         ( seperator )
 11          C@ 4 .R          ( display x-phase )
 12          DECIMAL          ( back to decimal )
 13          STEP-TIME @      ( get timer value )
 14          4 SPACES U. ;    ( and display it )
 15 ;S


Screen # 4
  0 ( The FORTH Step      3 of 10    02JUN82MBP )
  1
  2 : CTABLE <BUILDS  DOES> + ;
  3
  4 HEX CTABLE PHASES
  5  05 C, 09 C, 0A C, 06 C,          ( X-Forward )
  6  0A C, 09 C, 05 C, 06 C,          ( X-Reverse )
  7  06 C, 06 C, 06 C, 06 C,          ( X-Stop )
  8
  9  50 C, 90 C, A0 C, 60 C,          ( Y-Forward )
 10  A0 C, 90 C, 50 C, 60 C,          ( Y-Reverse )
 11  60 C, 60 C, 60 C, 60 C,          ( Y-Stop )
 12 DECIMAL
 13
 14 ;S
 15
```

Listing continued on next page

Older driver designs also produce the appropriate signals to the motor, but need the computer to send the driver two signals: a step signal that turns the motor one step, and a direction signal that tells the driver which way to turn the motor. One such driver chip is the SAA1027 by AIRPAX/North American Phillips and Signetics. (A FORTH system which communicates with this driver has been described in "Stepper Motor Control: a FORTH Approach," by Mark Bernstein in MICRO — The 6502/6809 Journal.)

Software Driver

Our approach in this example will be to control the stepper motor lines directly. In effect, we will build a driver chip in software.

The only interface between our computer's output ports and the motor itself will be an inexpensive transistor circuit designed to boost the milliamp of driving power at the port to handle the four amp load required to drive a single motor pole. The circuit is called a Darlington Array (see Fig. 1).

Eliminating an integrated circuit driver and taking direct control of the motor has certain advantages. For one thing, of course, we eliminate the cost of the chip, without making the circuitry any more complicated. Also we can, if needed, step the motor in half step increments for finer resolution (although we won't provide this capability in our example).

Perhaps the most important advantage is our ability to control acceleration or deceleration, which the chips do not provide. For heavier loads, the stepper motor must be accelerated gradually if it is to supply the necessary torque to get the load moving. Also, the motor can run at higher speeds. Trying to start a motor at too great a speed can cause it to stall. Our example will use an acceleration technique.

The Example

Our example controls two separate stepper motors. Together they control the x and y coordinates of some posi-

Continued on next page

```
Screen # 5
  0 ( The FORTH Step      4 of 10    02JUN82MBP )
  1
  2 : XY-RUN                  ( --- )
  3         ACCELERATOR       ( want to ramp up the speed )
  4         Y-PTR @ PHASES    ( y-addr )
  5         X-PTR @ PHASES    ( y-addr x-addr )
  6         4 0               ( 4 phases = 1 full step )
  7           DO
  8                 2DUP I + SWAP I +     ( get x,y offsets )
  9                 XY-MOTORS             ( x-addr y-addr --- )
 10           LOOP
 11         2DROP ;           ( or DROP DROP if you don't have it )
 12
 13 ( XY-RUN calculates the table address, then calls XY-MOTOR )
 14
 15 ;S


Screen # 6
  0 ( The FORTH Step      5 of 10    02JUN82MBP )
  1
  2 : X-STEP                          ( --- )
  3         X-COUNT @
  4           IF -1 X-COUNT +! X-DIR @
  5                IF 0 X-PTR ! 1 X-POSITION +!
  6                ELSE 4 X-PTR ! -1 X-POSITION +!
  7                ENDIF
  8           ELSE 8 X-PTR !
  9           ENDIF ;
 10
 11 ( X-STEP sets the forward or reverse bit pointer if there is )
 12 ( more steps to do and adjusts the position variable, else, )
 13 ( it sets the stop pointer. )
 14 ;S
 15


Screen # 7
  0 ( The FORTH Step      6 of 10    02JUN82MBP )
  1
  2 : Y-STEP                          ( --- )
  3         Y-COUNT @
  4           IF -1 Y-COUNT +! Y-DIR @
  5                IF 12 Y-PTR ! 1 Y-POSITION +!
  6                ELSE 16 Y-PTR ! -1 Y-POSITION +!
  7                THEN
  8           ELSE 20 Y-PTR !
  9           THEN ;
 10
 11 ( Y-STEP sets the forward or reverse bit pointer if there is )
 12 ( more steps to do and adjusts the position variable, else, )
 13 ( it sets the stop pointer. )
 14
 15 ;S


Screen # 8
  0 ( The FORTH Step      7 of 10    02JUN82MBP )
  1
  2 : XY-STEPPER                       ( --- )
  3         MAX-TIME STEP-TIME !       ( Setup for timer )
  4         Y-COUNT @ X-COUNT @        ( get x,y counts )
  5         MAX                        ( want the larger of the 2 )
  6         -DUP                       ( duplicate if not zero )
  7         IF 0 DO                    ( this is the number of loops )
  8                 X-STEP Y-STEP      ( get bit patterns )
  9                 XY-RUN             ( and step the motors )
 10              LOOP
 11         THEN ;
 12
 13 ( XY-STEPPER assumes the X & Y counts and direction flags )
 14 ( were set before entry )
 15 ;S
```

Listing continued on page 16

tioning device. An example application might be a drill press which must drill a series of holes in a circuit board. This routine will control the stepper motors that position the board under the drill bit, before each hole is drilled.

In this application, it isn't necessary to produce diagonals other than 45 degree angles (see Fig. 2). Because there is no time penalty, we've chosen to simply interlace the pulses to the two motors so that, when both are running, they run in step with one another. Any other approach would involve unnecessary complexity.

Since we are powering the coils of the motor directly, we need to supply the proper sequence of pulses. The table in Fig. 1 shows the four-step cycle of pulse patterns that will turn the motor one complete revolution (7.2 degrees). After the fourth pattern is output, the first pattern is sent again, and the cycle repeats.

These bit patterns are stored as bytes in a table called PHASES (Block 4 of the listing). The bit sequence in the first four bytes produces one forward-going cycle of the "X" motor. The next four bytes contain the same sequence in reverse. The final phase of both sequences is hex 6, which is phase 1, so that the motor will always stop at phase 1.

The next four bytes in the table PHASES each contain the phase-1 pattern. Cycling through these four bytes will cause the motor to stay at its stopping position.

The second half of the table contains the same bit patterns, but stored in the high-order nibble of the byte, where they will be output to the "Y" motor.

The four-byte sequence is cycled-through by the 4 0 DO . . . LOOP in Block 5. The appropriate sequence in the table is selected by the values in variables X-PTR (0 for forward, 4 for reverse, and 8 for stop) and in Y-PTR (12 for forward, 16 for reverse, and 20 for stop). These two variables are set up by the words X-STEP and Y-STEP respectively (Blocks 6 and 7). These words refer to the "count" and "direc-

Continued on page 16

FIG NATIONAL CONVENTION

October 9, 1982

RED LION INN/SAN JOSE
2050 Gateway Place, San Jose, CA 95110
9:00 a.m.-6:30 p.m./Conference and Exhibits • 7:30 p.m./Dinner and Speakers

FORTH APPLICATIONS

Learn about the latest FORTH applications and how to develop your own application programs.

Preregistration Form

Name(s) _____

Company _____

Address _____

City _____ State _____ Zip _____

Telephone _____ Ext. _____

☐ I am interested in presenting a paper on:

_____

☐ Enclosed is a check for:

#_____@ $5.00 registration(s) $_____

#_____@ $22.50 dinner(s) Reservations must be received before October 1, 1982. $_____

#_____@ $135.00 FORTH Vendor: 8'x10' exhibit booth(s) with draped table and 500 watts power. (Space limited, reserve early.) $_____

TOTAL $_____

Return to: FORTH Interest Group • P.O. Box 1105 • San Carlos, CA 94070

SPECIAL HOTEL RATES for Convention Attendees

Special hotel rates are available for attendees at the fourth FORTH Interest Group National Convention at the newly opened Red Lion Inn/San Jose. The special room rates may be obtained by telling the Red Lion Inn reservation desk that you will be attending the Convention. A room with two queen beds is $60.00 for one person or $70.00 for two persons. Alternately a room with one king bed may be reserved at the same rates. Write to the Red Lion Inn/San Jose, 2050 Gateway Place, San Jose, CA 95110 or telephone directly by calling (408) 279-0600.

The Red Lion Inn/San Jose is located near the San Jose Airport and has courtesy car service. Notification of your flight number, carrier, and time of arrival will help with scheduling airport pick up.

The FORTH Interest Group National Convention conference program and exhibitor displays are all scheduled at the Red Lion Inn/San Jose, making it the most convenient hotel for attending the convention.

tion" variables X-COUNT, Y-COUNT, X-DIR, and Y-DIR, to plug the correct values into X-PTR and Y-PTR.

The word XY-STEPPER (Block 8) defines a DO . . . LOOP that will run as many times as the greater of X-COUNT or Y-COUNT, to step the motors the right number of times. Inside the loop, X-STEP and Y-STEP check to see if the "x" or "y" count has gone to zero, and if so, stop the appropriate motor (refer back to Fig. 2).

The final three blocks put it all together to create a word called GOTO-XY, which uses the current position and the desired position to calculate the appropriate number of steps to move for each motor, then calls XY-STEPPER. For example, GOTO-XY could be given the coordinates for the position of a hole to be drilled in the circuit board. The stepper software will do all the rest.

Martin Petri is a consultant specializing in application software, hardware design, manufacturing and marketing. His most recent product is the Controlex CS-105 which he deomonstrated at the June 26, 1982 fig-FORTH meeting in Palo Alto.

Leo Brodie is Editor of FORTH Dimensions. □

```
Screen # 9
  0 ( The FORTH Step      8 of 10   02JUN82MBP )
  1
  2 : ?X-STEPS                  ( X-pos --- )
  3         X-POSITION @        ( get old count )
  4         -                   ( new-pos old-pos - )
  5         DUP
  6         0 <                 ( see if negative number )
  7           IF MINUS          ( get compliment )
  8             X-COUNT !       ( set the count )
  9             0 X-DIR !       ( set direction reverse )
 10           ELSE X-COUNT !    ( set the count )
 11             1 X-DIR !       ( set direction forward )
 12           THEN ;
 13
 14 ( ?X-STEPS calculates the # of steps and direction to move )
 15 ;S


Screen # 10
  0 ( The FORTH Step      9 of 10   02JUN82MBP )
  1
  2 : ?Y-STEPS                  ( Y-pos --- )
  3         Y-POSITION @        ( get old count )
  4         -                   ( new-pos old-pos - )
  5         DUP
  6         0 <                 ( see if negative number )
  7           IF MINUS          ( get compliment )
  8             Y-COUNT !       ( set the count )
  9             0 Y-DIR !       ( set direction reverse )
 10           ELSE Y-COUNT !    ( set the count )
 11             1 Y-DIR !       ( set direction forward )
 12           THEN ;
 13
 14 ( ?Y-STEPS calculates tne # of steps and direction to move )
 15 ;S


Screen # 11
  0 ( The FORTH Step     10 of 10   02JUN82MBP )
  1
  2 : GOTO-XY                   ( X-co-ord, Y-co-ord --- )
  3         ?Y-STEPS            ( calculate Y steps )
  4         ?X-STEPS            ( calculate X steps )
  5         XY-STEPPER ;        ( and move there )
  6 ( GOTO-XY is the high level word, expecting X & Y co-ordinates )
  7
  8 : INCHES 1000 * ;     : TENTHS 100 * ;     : HUNDREDTHS 10 * ;
  9 ( Conversion routines assumes 1 step = .001 inch )
 10
 11 : X-PLOT        0 Y-COUNT ! X-POSITION @ + ?X-STEPS XY-STEPPER ;
 12 : Y-PLOT        0 X-COUNT ! Y-POSITION @ + ?Y-STEPS XY-STEPPER ;
 13 ( Can be used in the form 5 INCHES X-PLOT or -5 TENTHS Y-PLOT )
 14
 15 ;S
```

End of listing

Handling Interrupts in FORTH

Stephen Melvin

When FORTH is used in a process control application, it is often desirable to have a high-level FORTH word execute in response to an externally generated interrupt. This article stemmed from a motor controller project and the desire to use FORTH for everything, including the interrupt service routine. The implementation shown in Figure 1 was used for the pulse width modulation control of a DC motor. An 8253 interval timer was the central element in the system. An interrupt occurred every 250 ms and the interrupting word updated the speed and provided feedback information to the main routine through a shared variable. FORTH was extremely helpful in debugging the system and undoubtedly saved much time. Furthermore, all of the software fit in three screens.

Although an interrupt processing scheme must by definition be hardware dependent, the ideas presented here are very general and can be applied to most FORTH systems.

Figure 1.

```
SCR #1
  0 ( INTERRUPT HANDLING )
  1 CODE RETURN  PSW POP H POP D POP B POP RET  END-CODE
  2 : INT  BELL RETURN ;
  3 CODE INTSRV  B PUSH D PUSH H PUSH PSW PUSH
  4              ' INT B LXI   NEXT JMP    END-CODE
  5 : PUT  [COMPILE] ' CFA ' INT ! ;
  6 : TEST  ' ;S INTSRV ;
  7 : SETINT  C3 OVER C! 1+ ' INTSRV SWAP ! ;
```

Theory of Operation

In the discussion that follows, the definitions below are used.

HIGH-LEVEL DEFINITION: A list of addresses of words to be executed. Each address actually points to the cfa of the word to be executed. Each cfa, in turn, is a pointer to executable code.

IP: The interpreter pointer. Points to an element inside a high-level definition; thus a pointer to a pointer to a pointer to executable code (yes, three levels of indirection).

NEXT: A routine that advances the IP and jumps to the code pointed to by the location pointed to by the location pointed to by the previous value of the IP.

DOCOL: The **DOES>** portion of : . A routine which loads the IP with a parameter field address (PFA) passed to it after first pushing the previous contents of the IP onto the return stack.

;S : The word compiled by ; . A routine which loads the IP with the value on top of the return stack. (*Editor's Note: In 79-Standard and Starting FORTH, this word is called* **EXIT**.)

EXECUTE: A routine which jumps to the location pointed to by the location pointed to by the value on the top of the data stack.

The processing of an interrupt in FORTH requires seven basic steps:

1) upon receipt of an interrupt, passing control to the interrupt service routine;
2) saving the state of the processor;
3) passing control to FORTH;
4) executing the FORTH word;
5) returning to the interrupt service routine;
6) restoring the state of the processor; and
7) returning to the routine that was interrupted.

The term "interrupting word" will be used to refer to the FORTH word which is intended to be executed upon receipt of the interrupt. Steps 1, 2, 6 and 7 are relatively simple and most microprocessors provide for handling them quite easily. Note, however, that the "state" of the processor may include external variables as well as internal registers. Generally, the saved information is pushed onto the system stack in step 2 and popped off the stack in step 6 (the CPU's stack may correspond to FORTH's data stack, its return stack or neither).

The main problem comes with steps 3 and 5. There are several different ways to handle them as there are many potential entry points into FORTH. All methods, however, must sooner or later load the IP and they can be classified according to how it is done. There are three categories of methods as follows:

1) those which initialize the IP directly (i.e., from assembly language);
2) those which depend on DOCOL to initialize the IP; and
3) those which depend on **;S** to initialize the IP.

Note that a particular method may fall into one of these categories indirectly (for example, if **EXECUTE** is jumped to without first initializing the IP, then the method would fall into category 2 since it would have to jump to DOCOL to work). Those methods falling into category 2 have the disadvantage of having to clean up the garbage on the top of the return stack and may have to deal with passing the PFA to DOCOL. The methods of category 3 are sometimes even more involved since some assemblers don't provide easy access to the return stack. So, category 1 seems to be favorable since loading the IP directly shouldn't present a problem. However, there are three basic methods *within* group 1. They are:

1) loading the IP and jumping to **EXECUTE**;
2) loading the IP and jumping to DOCOL; and
3) loading the IP and jumping to NEXT.

Continued on next page

If **EXECUTE** is used, then the top of the data stack must be initialized. Furthermore, to use DOCOL would require a PFA (or something to be treated like a PFA) to be passed. However, the third method doesn't require any other initialization.

Now the question arises of passing control back to the "restore and return" routine (steps 6 and 7). To do this, a pointer to a pointer to the executable code must be left somewhere that the IP will point to after the interrupting word is finished. The easiest way to do this is simply to set up a place in the dictionary with the interrupting word's code field address (CFA) followed by the return routine's CFA (recall that a CFA is a pointer to a code field, which points to executable code). Then, by initializing the IP to the address of the first location and jumping to NEXT, execution will naturally continue with the return routine after the interrupting word is done.

A final consideration of the interrupt processing scheme is how to achieve goal #2 (the ability to reassign the interrupting word). Since we have to reserve a location in memory for the word's CFA, the obvious approach would be to allow the CFA of another word to be stored there.

Specific Implementation Details

This section provides an example of the method described above for an 8080-based FORTH system with suggestions for systems based on other processors. As illustrated in Figure 1, the complete implementation is fairly short and has little assembly language.

The first question is: what information is saved by the processor when an interrupt occurs? Some CPUs (e.g., 8080, Z80 and 2650) push just the program counter (PC) onto the stack, others (e.g., 6502 and Z8) save the PC and a status register while others still (e.g., 6800, 6809) save the PC and *all* registers. The code on line three of Figure 1 is what is necessary to save the information which has not already been saved. Note that if the IP and the so-called W registers do not reside in the processor, then they must be retrieved from memory and saved. Also note that stack pushes need not be used. For example, a Z80 user might want to perform **EX AF,AF'** and **EXX** to switch register banks or a Z8 user might want to simply reload the register pointer register.

The next task is to load the IP (in this example, the IP is stored in register pair BC of the 8080). The IP is loaded with the PFA of **INT** (which is a pointer to the CFA of the interrupting word [initialized to **BELL**]). Then, there is a jump to NEXT which will cause the interrupting word to be executed by the FORTH system. Finally, **RETURN** will be executed, which is the opposite of line three followed by a return instruction.

Note that the CFA and the terminating **;S** of **INT** are never used and to save space they could be eliminated (for example, if the ; in line two is replaced by **[SMUDGE** then no **;S** will be compiled into the dictionary). In fact, the header could be eliminated also but retaining it makes things easier by allowing ' to be used.

PUT is fairly straightforward. It simply stores the CFA of a new word into the location pointed to by the PFA of **INT**. See Figure 2 for an example of how it is used. The word **TEST** simulates an interrupt and is hardware dependent. It should be rewritten to reflect what an interrupt does, pretending that the machine is executing **;S**. In this example, the PFA of **;S** is pushed onto the data stack (which in this case is the same as the 8080's stack) in order to fake a PC push. Then, **INTSRV** is jumped to. **TEST** can be entered directly from the terminal to test the system (see Figure 2).

SETINT is specific to the 8080. It puts a jump to the PFA of **INTSRV** at a certain location in memory. For example, typing **20 SETINT** would set up for a RST 4 instruction. **SETINT** must be modified to accommodate whatever method is used to define the interrupt service address.

Implementation of Multiple Interrupts

Consider the problem of handling multiple interrupts in FORTH. The most direct way would simply be to define a new **INT**, **INTSRV**, **PUT**, **TEST** AND **SETINT** for each interrupt (only one **RETURN** would be required). This approach would be quite reasonable for a small number of interrupts since **INT** and **INTSRV** are relatively short and the other three words would probably only be needed for debugging. In fact, any other kind of multiple interrupt processing scheme would cost additional speed. However, if space is scarce and speed isn't too important, another method could be adopted. The type of method used would depend heavily on the hardware but the general idea would be to be able to have only one interrupt service routine. One such method which used a look-up table was implemented on an 8080-based system but it was not found to be very useful. If any readers have come up with multiple interrupt routines that they like, please send them in.

This article has attempted to provide a basis for interrupt handling in FORTH which can be applied to most systems with some basic knowledge of the particular configuration. However, if there were considerations which were neglected then please let us know. Also, there must be many other interesting applications of interrupts in FORTH so send them in!

(Editor's Note: This code was written for fig-FORTH. To run the code on 79-Standard or Starting FORTH implementations, change ' to **[']**, *except in line 5 change the phrase* **[COMPILE] '** *to* **'** . *Change* **;S** *to* **EXIT**.*)*

Stephen Melvin is a graduating senior in Electrical Engineering at the University of California at Berkeley. □

Figure 2.

```
1 LOAD OK
: STAR 42 EMIT ; OK
: STARS  10 0 DO STAR LOOP ; OK
PUT STAR OK
TEST *OK
PUT STARS OK
TEST **********OK
PUT CR OK
TEST
OK
```

Towards a New Standard

Robert L. Smith

The Standardization Effort

The FORTH Standards Team met in May at the National 4-H Center, Washington, D.C. The team decided to work towards a new FORTH Standard, tentatively called FORTH-83. The mechanism of producing a FORTH Standard seems to be evolving. Previously the team members met, discussed, and then voted on a variety of topics during a three day session. Ambiguities and smooth wording were worked out by a smaller group of referees. The resulting document was then offered for acceptance as a whole by at least two-thirds of the voting members. One of the problems with this method is that the time for deliberation is far too short for the proposals and implications to be thoroughly understood.

The next Standard will evolve through various working drafts. It is intended that there be opportunities for public examination and input. By having more than one meeting prior to acceptance of the next Standard, changes and corrections can be made which should reduce the inconsistencies in the final document.

Perhaps the future standardization efforts should be split into separate functions somewhat like that of the COBOL or MUMPS standardization committees. At the lowest level a language development committee meets regularly to make changes to the language. Their output is published as a journal, for consideration and testing by implementers and users. A separate Standards committee generates an actual Proposed Standard document on the time scale of five years. They freeze the output of the language development committee (who continue to work independently). After a suitable voting process, this document becomes the new Standard. Thus the community of users has an adequate chance to make their views known. Additional steps in the process involve approval by an ANSI committee, then perhaps other governmental or quasi-governmental committees.

Vocabularies

The area of greatest concern for the next Standard is that of vocabularies. FORTH-79 has a very weak vocabulary structure. It was chosen as the minimum subset of most FORTH implementations. The only weaker structure is the complete lack of vocabularies in older versions of FORTH, such as DECUS or OVRO (Cal Tech) FORTH. In FORTH-79, the search order at a given time is through two vocabularies: the one specified by CONTEXT and then the FORTH vocabulary. Some other FORTH systems (like fig-FORTH) have vocabularies linked together in a tree structure determined when the vocabularies are created. The search order is determined by the vocabulary last activated and its predecessors in the tree structure down to the trunk of the tree (which is usually FORTH). In poly FORTH systems the search order when a given vocabulary name is invoked is determined by a four nybble (in one word) parameter given when the vocabulary name is created. Typically this limits the total number of separate vocabularies to 7 or possibly 15.

A dynamic method for determining search order uses the "vocabulary stack." This is a concept taken from STOIC. Each wordset is "sealed," i.e., not linked to any other. A wordset is pushed onto the vocabulary stack from, say, the value in CONTEXT by using a word such as VPUSH (my favorite name for this function is ALSO). Another word is used to drop the top member of the vocabulary stack, or perhaps to clear it out entirely. Bill Ragsdale uses the word ONLY for the latter purpose. By first searching CONTEXT and then the vocabulary stack we can maintain a reasonable amount of upward compatibility. This is an idea advanced by George Shaw at the last FORML Conference.

There are many other possibilities. Don Colburn has suggested a defining word like SEARCH-ORDER which would name a word which specifies the search order. John James has suggested that the invocation of a vocabulary or wordset name would push itself onto the vocabulary stack if it was not currently on the vocabulary stack. Otherwise the stack would be truncated back to its first appearance on the stack.

There are other designs for vocabulary mechanisms. Almost any of them would be an improvement over FORTH-79. In my opinion it is important that the next Standard have a significant improvement in vocabulary structures. If you have any strong opinions on this matter, please communicate them in writing to the FORTH Standards Team.

FORTH Standards Team Upcoming Working Meeting

A working session in the development of the FORTH language 1983 standard (FORTH-83) has been scheduled this October 3rd through 5th in Carmel Valley, California. Space will be limited with priority for existing standards team members. Accommodations will cost US$150 based on double occupancy including meals. Room reservations require a deposit of US$50 and should be received by July 31.

This working session will attempt to resolve the FORTH-83 Standard working draft in anticipation of an accepted standard near the beginning of 1983. This working draft will be available for US$15 beginning August 1. Comments on this working draft are encouraged. Standards team sponsors additionally receive all mailings to team members prior to the October meeting, including copies of submitted proposals and comments. Standards team sponsorship is available for US$50.

Please send orders, deposits or inquiries directly to the FORTH Standards Team, P.O. Box 4545, Mountain View, CA 94040, USA; or telephone Mr. Roy Martens at (415) 962-8653.

Defining Words II

Henry Laxen

Last time we took a look at defining words and went through two simple examples. One was a defining word that sent strings of characters to the terminal, and the other one gave suitable responses to simple commands. This time I will continue on the theme of defining words, and look at a more meaty example. We will construct a defining word that constructs defining words. Try saying that 5 times real fast backwards.

One of the problems with many of the published examples on defining words is that they seem to be trivial at first glance. You get the feeling that you are being cheated when you see a 1 line definition of ARRAY or VECTOR. After all, we know how complicated arrays and vectors are and it doesn't seem possible that they can be implemented so trivially in FORTH. Well, today's example is a little rougher, but if you just keep a few simple principles in mind it will be easy to follow. In a nutshell, defining words consist of:

1. The word **CREATE**, which when executed takes the next word in the input stream and makes a dictionary entry for it. The word thus created is called a member word.
2. The words between **CREATE** and **DOES>** which specify the *compile time* behavior of the defining word. These execute when the word containing the **CREATE** and the **DOES>** executes.
3. The words between **DOES>** and ; which specify the *run time* behavior of the member words. These execute when the word defined by the **CREATE DOES>** pair is executed. As a bonus, before the words between the **DOES>** and the ; are executed, the parameter field address (pfa) of the member word is pushed onto the parameter stack.

Remember these three facts, and apply them whenever you see a defining word. They completely describe the compile and run time behavior of the FORTH machine. If the above aren't clear to you, I suggest you reread the defining words article printed in the last issue of *FORTH Dimensions*, and apply the rules to the examples presented. We will now apply them to a new example.

The problem we want to solve is to be able to create classes of items, and ask questions about whether an item belongs to a given class or not. We would also like to know whether or not two items are the same or not. Notice we have two levels of definition going on here. The first level is the name of the class, and the second is the individual elements of the class. For example, the name of a class could be **COLOR** and the elements of the class **COLOR** could be **RED**, **WHITE**, and **BLUE**. One way to implement this is to create a defining word, called **CLASS** which creates classes of things, such as **COLOR**. Now **COLOR** itself can be another defining word which creates items belonging to its own **CLASS**. Only one question remains before we can implement the **CLASS** concept, and that is the matter of representation. There are several ways we could indicate that an item is a member of a **CLASS**, for example each class could define an array, with items belonging to the class being elements of the array. Another approach is to store a **CLASS** identifier with each element of the class. The approach we will take is to create a linked list of items belonging to a particular class. The advantages of this approach are that we do not need to specify the size of the class at compile time, we do not need to store redundant information, and it is very easy to implement.

Now let's take a look at the code in Block 24. We will represent each class by a unique class number, and each


```
    24
 0 \ Set Theory
 1 VARIABLE ITEM#     0 ITEM# !
 2 VARIABLE CLASS#    0 CLASS# !
 3 VARIABLE NULL      0 NULL !
 4 : LINK   (S addr -- )
 5    HERE   OVER @ ,   SWAP !   ;
 6 : CLASS
 7    CREATE   0 , ( LINK )   CLASS# @ ,   1 CLASS# +!
 8    DOES>
 9       CREATE   LINK   ITEM# @ ,   1 ITEM# +!
10       DOES> 2+ @   ;
11 : MEMBER?   (S n pfa -- f )
12    0 -ROT   BEGIN   @ DUP WHILE   2DUP 2+ @
13    = IF   2DROP DROP 1 1 NULL   THEN   REPEAT   2DROP   ;
14 : MEMBERS   (S pfa -- )
15    BEGIN   @ DUP WHILE   DUP NFA ID.   REPEAT   DROP   ;
```

```
    149
16JUN82HHL \ Set Theory                                  16JUN82HHL
          ITEM#      Uniquely identifies the name as an item number
          CLASS#     Uniquely identifies the class as a class number
          NULL       Always zero, for finishing a linked search
          LINK   (S addr -- )
            Add a link to a linked list chain. Addr is the head.
          CLASS
            A defining word that creates defining words.  A unique
            class number is assigned to a class.  The class name can
            be used to create items belonging to the class.  An item
            number is assigned to each member.  Members are linked.
          MEMBER?   (S n pfa -- f )
            Takes an item number and a pointer to a class, and returns
            True if the item is a member of the class.  Else False.
          MEMBERS   (S pfa -- )
            Takes a pointer to a class and lists its members.
```

item by a unique item number. The word **LINK** is used to add a new item to a linked list. The address it expects on the stack contains the address of the head of the list. Let's look at **LINK** in detail. Suppose that **HERE** is at 17000, then:

```
addresses      contents      15000 LINK [CR]    contents
  10000          0                                0
  12000          10000                            10000
  15000          12000                            17000
  17000          ?????                            12000

               Stack                          Location 15000   Location 17000
15000 LINK     15000                            12000            ?????
  HERE         15000  17000                     12000            ?????
  OVER         15000  17000  15000              12000            ?????
  @            15000  17000  12000              12000            ?????
  ,            15000  17000                     12000            12000
  SWAP         17000  15000                     12000            12000
  !            Empty                            17000            12000
  ;
```

Thus after the execution of **LINK**, location 15000 now points to the new head of the linked list, namely 17000, which now points to the previous head of the linked list, namely 12000. Also, after **LINK** has executed, **HERE** would be at 17002, due to the , that was executed as part of **LINK**. We have just created what is called a singly linked list; that is, a list where the pointers only run in one direction. As an exercise, you might try to create a doubly linked list, that is, a list where pointers point to the next and the previous element. For extra credit, figure out how to create a double linked list using only a single cell for the pointer!!

Now then, let's continue on with our **CLASS** example. We will apply our three rules to the definition of **CLASS**.

1. We see a **CREATE**, so we know that **CLASS** is a defining word. Thus when **CLASS** is used, we expect it to be followed by a yet undefined word which it will define, such as **COLOR**. **COLOR** would be called a member word of the defining word **CLASS**.

2. When **CLASS** is executed, the words between **CREATE** and **DOES>** are executed, determining the compile time behavior of the word **CLASS**. They do the following. First the **ZERO** , is executed, which initializes the linked list to empty. Next the class number is stored and incremented. That is all that happens at compile time.

3. When the word defined by **CLASS**, such as **COLOR**, is executed, the parameter field address of **COLOR** is pushed onto the parameter stack, and then the words between the **DOES>** on line 8 and the ; on line 10 are executed. (Question: What is the significance of the parameter field address of **COLOR**? Answer: It contains the pointer to the head of the item list, which was initialized to **ZERO** and is followed in the next cell by the class number of this class. Thus the parameter field address allows us to uniquely identify the **CLASS** of **COLOR**, as opposed to some other class.)

Now then let's look at what happens when **COLOR** is executed. We start with the pfa of **COLOR** on the stack. Next we encounter a **CREATE**. That means we are executing a defining word!! It looks like we'll have to apply rules 1-3 one more time, but this time they describe the behavior of **COLOR**, not of **CLASS**.

1. The word **CREATE** is executed, and it gets the next word in the input stream and adds it to the dictionary. Thus we expect **COLOR** to be followed by another word, say **RED**. Note the pfa of **COLOR** is still on the parameter stack.

2. The words between the **CREATE** and the **DOES>** are executed, and determine the compile time behavior of the word **COLOR**. The first thing they do is **LINK** up the pfa of the member word, **RED**, to the pfa of the defining word **COLOR**. Next the Item number is stored and incremented. That is all that happens at the compile time of **COLOR**, since the next word we see is **DOES>**.

Continued on next page

```
     25
 0 \ Define some classes                                  16JUN82HHL
 1 CLASS COLOR
 2    COLOR RED   COLOR WHITE   COLOR BLUE   COLOR GREEN
 3    COLOR YELLOW
 4 CLASS TEXTURE
 5    TEXTURE HARD   TEXTURE SOFT   TEXTURE ROUGH
 6    TEXTURE SMOOTH
 7 YELLOW ' COLOR MEMBER? ,
 8 BLUE ' COLOR MEMBER? ,
 9 HARD ' COLOR MEMBER? ,
10 ROUGH ' TEXTURE MEMBER? ,
11 WHITE ' TEXTURE MEMBER? ,
12 HARD WHITE = ,
13 GREEN BLUE = ,
14 SOFT SOFT = ,
```

3. When the word defined by **COLOR**, say **RED**, is executed, we push the parameter field address of **RED** on the stack, and execute the code between the **DOES>** and the ; on line 10. It simply increments this pfa by 2 and fetches the contents of that location. In Step 2 just above we saw that we stored the item number corresponding to **RED** in that location. Thus the run time behavior of an element of a class is to push its item number on the stack.

We have succeeded in creating a defining word, called **CLASS**, which in turn defines more defining words. Look at Block 25 to see how it can be used. We use **CLASS** to define the classes of **COLOR** and **TEXTURE**. Next we use the word **COLOR** to define some members of its class, such as **RED**, **WHITE**, and **BLUE**. Furthermore we use the word **TEXTURE** to define some members of its class, such as **HARD** and **SOFT**. Now we want to be able to ask questions like, is **YELLOW** a member of the class **COLOR** or is **WHITE** a member of the class **TEXTURE**? To answer this question we define a word called **MEMBER?** on lines 11 thru 13 of Block 24. It expects an item number and a pointer to a **CLASS** on the stack. It returns a flag indicating whether the item number is a member of the class. We can get an item number by simply executing the name of an item, such as **RED** or **SMOOTH**. It's a little trickier to get a pointer to a class, since when we execute a class name we define a new word, and we don't get a pointer. We solve this problem by using ' (tick) to get a pointer to a class. Thus to determine if **YELLOW** is a member of **COLOR** we would have to type:

YELLOW ' COLOR MEMBER? [CR]

MEMBER? is defined to run through the linked list and see if the item numbers match. If they do, it returns true, otherwise it returns false. Notice the use of the **NULL** variable to terminate the search in the event of a match. The search continues until the zero link is found, however if we get a match, we want to terminate immediately. We arrange this by throwing away the pointer we have been chasing and replacing it with a pointer to a zero, which is exactly what **NULL** provides.

Finally, on lines 14-15 of Block 24 we define the word **MEMBERS** which lists all of the members of a particular class. It too expects a pointer to a class on the stack, which must be provided by ' .

Next time we'll finish up our series on defining words by generalizing the concept of **CLASS** to an arbitrary number of levels, and by combining it with recursion to create "n"-way trees and other linked structures. Stay tuned to this station for all the details. That's all for now, good luck, and may the FORTH be with you.

Henry Laxen is part of Laxen & Harris Inc., a FORTH teaching and consulting company based in Hayward, California. □

Source Screen Documentation Tool

Kim Harris
Laxen & Harris, Inc.

This article describes a simple but powerful program which extracts and prints information from FORTH source screens. This tool will print lines of a screen which start with a non-blank in the first column. It also prints the screen number and line 0. If source screens are entered according to a simple, natural FORTH style, then this tool extracts most of their useful information.

Two words are defined for the tool:

1OUTLINE #screen –
Print "outline" of one block. Display the #screen, line zero, and any lines containing a non-blank in the first column.

OUTLINE #1st-screen #last-screen –
Print lines as described in 1OUTLINE above for the range of screens from #1st-screen through #last-screen. It is used like INDEX.

An example of using this tool on its own source screen is shown below. The first line of the compressed "listing" shows the screen number and line 0.

The next four lines are indented and are copies of lines 2, 3, 11, and 12 of the source screen. These lines have a non-blank in the first column. Lines with a blank in the first column are ignored.

This tool works best if a particular style for the layout of a source screen is used. The style is simple to remember and fits well with normal Forth programming. The goal is to permit **OUTLINE** to find the words defined in a screen and selected comments.

1) Overall format:
   a) Line 0 should be used for a comment only.
   b) Line 15 should be left blank.
   c) Any line may have text from the first column UP TO the last column. The last column of each line should be blank to avoid running into the start of the next line.
2) Start each definition in the first column of a new line. This applies to colon definitions and all other classes (e.g., **CONSTANT**s, **VARIABLE**s, etc.)
3) Follow each defined name by a "stack comment" similar to that of a glossary entry.
4) Follow each stack comment by a "purpose comment" (i.e., WHAT the word does, not HOW it does it). If the comment is continued to the next line, put an open parenthesis in the first column of the second line. Although this is not required for FORTH, it allows **OUTLINE** to pick up this comment line.

The documentation tool is compatible with the following dialects (except as noted below): 79-STANDARD, Starting FORTH, fig-FORTH, and polyFORTH.

The implementation assumes 1024 byte disk buffers and 64 byte lines for screens. On a single-user FORTH system, **TYPE** may be used instead of **>TYPE**. On a multi-user system, **>TYPE** is needed to avoid conflicts between shared disk buffers and interrupt controlled Input/Output operations. **>TYPE** copies its string to PAD before typing it.

```
Screen # 155
  0 ( Source screen documentation tool                  WF25 2Apr82 KRH )
  1
  2 : 1OUTLINE    ( #screen -- ) ( print line 0 & lines with
  3 (  non-blank in column 1 )
  4                       CR CR  ." Screen # "  DUP .    1024 0  DO
  5              DUP BLOCK    I + DUP C@  BL = NOT    I 0=    OR IF
  6                     3 SPACES    64 -TRAILING )TYPE    CR    ELSE
  7                                                  DROP    THEN
  8                                                          64 +LOOP
  9    DROP    ;
 10
 11 : OUTLINE    ( #1st-screen #last-screen -- ) ( print line 0 &
 12 (  lines with non-blank in column 1 for range of screens )
 13    1+ SWAP DO    I 1OUTLINE    LOOP    ;
 14
```

Above is the source for this documentation tool. Saying **155 1OUTLINE** produces the "listing" below.

```
Screen # 155     ( Source screen documentation tool                  WF25 2Apr82 KRH )
   : 1OUTLINE    ( #screen -- ) ( print line 0 & lines with
   (  non-blank in column 1 )
   : OUTLINE    ( #1st-screen #last-screen -- ) ( print line 0 &
   (  lines with non-blank in column 1 for range of screens )
```

The Art of Recursion

Bob Gotsch

Inspired by the interesting "Roundtable on Recursion" in Vol. III, No. 6, FORTH Dimensions, I would like to offer some models for recursive style in FORTH. Several kinds of recursion are possible in FORTH, so long as stack limits are recognized. Whether to use recursion at all may be questioned by some; it has the distinction of being much more interesting — and slower — than branching and iteration with conventional control structures.

To begin, two words are needed, **MYSELF** which calls the word-as-a-whole in which it is used, and **RETURN**, which is simply

```
: RETURN    R> DROP ;
```

(Editor's note: The 79-Standard word **EXIT** *has the same function.)* **RETURN** is used in a "stoprule" which stops and returns execution to the calling level. Remember, when **RETURN**ing, FORTH lets you decide whether to **DROP** local variables associated with that instance of the procedure. The following procedure **COUNTUPTO** illustrates this.

```
: COUNTUPTO (level# ---)
  DUP 0= IF DROP RETURN THEN (stoprule)
    DUP 1 -        (put 1 param on stack)
      MYSELF (level# ---)
    CR . " LEVEL "
    . ;          (remove 1 stack param)
```

The procedure-as-a-whole destroys the parameter it starts with, so **MYSELF** must behave the same way.

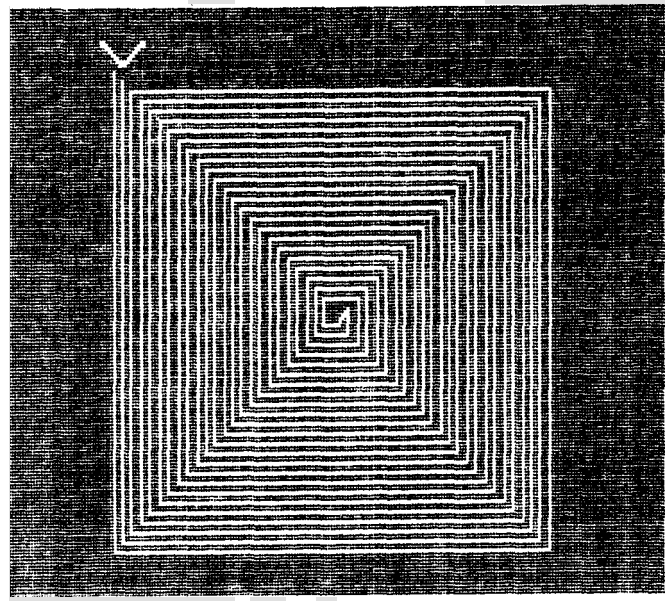
If there is a risk of exceeding data stack capacity (93 in the FORTH I use), an alternative to piling up so many local variables is to have the procedure reverse the parameter change it makes. **PROCEDURE.IN&OUT** decrements by 3 before calling itself and increments by 3 upon **RETURN**ing. Also it draws a diagram of its own action.

```
0 VARIABLE LIMIT

: PROCEDURE.IN&OUT (size --- size+6)
  DUP LIMIT @ <
  IF 135 RIGHT 4 FORWARD 135 LEFT 6 +
  RETURN THEN              (stoprule)
    DUP FORWARD
    90 LEFT
      3 - (change stack for next level)
        MYSELF (size --- size+6)
      3 + (reverse change upon return)
    90 RIGHT
    DUP BACKWARD ;
```

If you will accept as self-evident the relative turning and displacement commands **FORWARD**, **BACKWARD**, **RIGHT**, and **LEFT**, each consuming one parameter, here is the figure it draws.

(Editor's note: The author is using the syntax of "turtle graphics," where lines are drawn as though they were tracks left on the screen by a cursor [called the "turtle"]. The commands used to move the cursor are these:

n FORWARD	*Moves the cursor n units forward in the direction it is already pointing.*
n BACKWARD	*Negates n, then calls* **FORWARD.**
n RIGHT	*Changes direction by adding n [in degrees] to the current heading.*
n LEFT	*Negates n, then calls* **RIGHT.**)



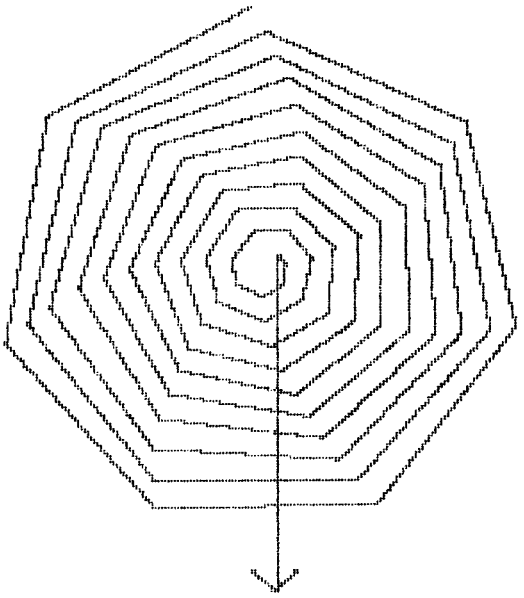
A minor disadvantage is that as **MYSELF** had to leave a value on the stack for each next level coming back up, so does the procedure-as-a-whole leave a value. A named variable could have been used instead, or in this example a "setup word" could also do stack cleanup.

```
: IN&OUT (startsize, limit ---) LIMIT !
  GRAPHINIT POSITION ARROWHEAD
  PROCEDURE.IN&OUT DROP ;
```

This variation minimizes use of data stack but could reach return stack limits (112 in the FORTH I use). Some practical applications of recursion, with branching, have more "lateral" activity rather than going to such depth. I will **RETURN** to branching, with examples, a little later. But another way to avoid return stack limits is to write a procedure that does its work "as it goes" rather than "as it comes back" up the levels. If it invokes itself repeatedly as the last instruction, without coming back, it is called "tail recursion" or sometimes "not true recursion." Some folks even claim that the job can be done more efficiently with iterative structures such as **DO LOOP** and **BEGIN AGAIN**. Probably such practical people will not see any point in this article-as-a-whole.

Here is a simple example that draws a diagram of "tail recursion," jumping out of the cycle when a limit condition is reached.

```
: CYCLEUNTIL (startsize ---)
  DUP 5 <
  IF 90 RIGHT 95 + FORWARD ARROWHEAD
  RETURN THEN       (stop rule)
    6 LEFT DUP FORWARD
    1 -
    R> DROP MYSELF ;
```



Notice the same form, a "stoprule" at the beginning of the procedure. **R> DROP** prevents useless build-up on the return stack, allowing unbounded repetition. But if **CYCLEUNTIL** were called from within another word, the **R> DROP** first removes from the return stack the address of the next action in the "calling word," jumping over, instead to the level that called the "calling word." To avoid this little inconvenience I propose to have the recursive word push its own PFA on the return stack with a variant of **MYSELF** called **TAILMYSELF**.

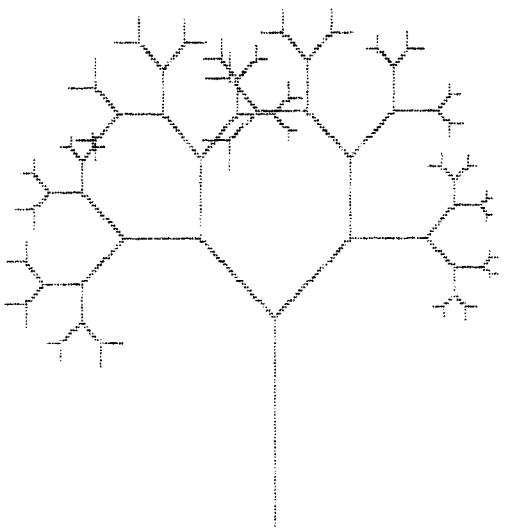
```
: TAILMYSELF (---)
  LATEST PFA [COMPILE] LITERAL
  COMPILE >R ; IMMEDIATE
```

I show its use here in a division-by-subtraction operation that ACCUMULATEs the quotient in a variable of the same name. You don't have to tell me that **/MOD** is available to do the same thing at a saving of 30 seconds. Note the recursive call is the last word in the procedure.

```
0 VARIABLE QUOTIENT

: ACCUMULATE (divdnd, divsr --- remndr)
  2DUP <
    IF DROP RETURN THEN       (stoprule)
      1 QUOTIENT +!
      2DUP - ROT DROP SWAP
  TAILMYSELF (divdnd, divsr --- remndr)
;

: DIVISION (dividend, divisor ---)
  0 QUOTIENT !
  ACCUMULATE
  CR ." QUOTIENT IS " QUOTIENT ?
     ." REMAINDER IS " . ;
```

Returning to "depth recursion," that **RETURN**s in order to branch, this next procedure draws a branching tree. **PENDOWN** or **PENUP** determine whether the moving "turtle" draws or not. **BI-TREE** includes two recursive calls, one for each side of the "V." The "Y-shaped" element spans two levels. Comparing with **IN&OUT** above, this procedure returns with the parameter still on the stack — for use on the other branch. In stepping thru, trust that the procedure-as-a-whole is transparent and ignore the **MYSELF**s. In fact the Zen of reading or writing recursive procedures is to be NOT distracted to other levels by the **MYSELF**. Just be in this instance.

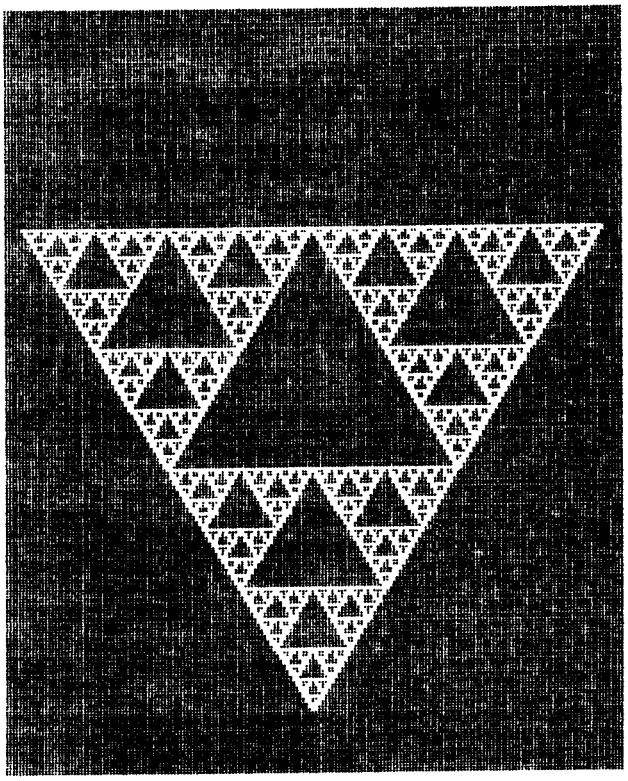
```
0 VARIABLE LEVEL

: BI-TREE (size --- size)
  LEVEL @ 1 < IF RETURN THEN (stoprule)
  PENDOWN DUP FORWARD
    -1 LEVEL +!
    45 LEFT
    DUP 6 9 RANGERAND 11 */ (branchsiz)
      MYSELF (size --- size)
    90 RIGHT
      MYSELF (size --- size)
    DROP          (branchsiz)
    45 LEFT
    1 LEVEL +!
  PENUP DUP BACKWARD ;

: SETUP.BI-TREE (size,level---) 2 ENUF?
  LEVEL !  GRAPHINIT FULLSCREEN
  PENUP  95 BACKWARD  BI-TREE  DROP ;
```



Continued on next page

In one last example that parallels the first, parameters are piled on the stack, to be used for drawing on the way back up the levels, after the stoprule has been encountered. The pattern was drawn with starting size of 180, so it's somewhat possible to see why there are six sizes of triangles. Excuse the mixing of iteration and recursion. It just seemed more readable that way.

```
: HALVE   2 / ;          : TIMES   0 ;
: PROC.NESTED-TRI (size ---)
  DUP 5 <
    IF DROP RETURN THEN      (stoprule)
    3 TIMES DO
      DUP HALVE
      MYSELF (size ---)
      DUP FORWARD
      120 RIGHT
    LOOP
  DROP ;
```



Bob Gotsch is a graphics programmer for Time Arts, Inc. and a teacher of graphic arts at California College of Arts and Crafts. He's interested in exploring the use of computers as aids to artists. □

FPA FORTH PROGRAMMING AIDS by Curry Associates

FORTH PROGRAMMING AIDS is a software package containing high level FORTH routines which allow the FORTH programmer to write more efficient programs more quickly, and they are especially useful for those who are using a metacompiler or cross compiler.
FORTH PROGRAMMING AIDS allow the programmer to

- Minimize memory requirements for target systems by finding only those words used in the target application.
- Tailor existing words (including nucleus words) to specific needs by decompiling the word to disk, editing, and recompiling.
- Build on previous work by transferring debugged FORTH routines (including constants and variables) from one application to another.
- Speed program loops by finding **all** words called from the loop for possible merging or recoding to assembler.
- Patch changes into compiled words in seconds.

FORTH PROGRAMMING AIDS comes with complete source code and a 40-page manual. FPA contains the following modules:

DECOMPILER This is a **true** decompiler which converts the FORTH words in RAM into compilable, structured FORTH source code, including program control words such as IF, ELSE, THEN, BEGIN, etc. If you ask FPA to DECOMPILE the nucleus word INTERPRET, you get the following output displayed on your terminal within 3 seconds:

```
( NFA&PFA: 4094 4108 )
  : INTERPRET
       BEGIN -FIND
          IF STATE @ <
             IF CFA ,
             ELSE CFA EXECUTE
             THEN ?STACK
          ELSE HERE NUMBER DPL @ 1+
             IF [COMPILE] DLITERAL
             ELSE DROP [COMPILE] LITERAL
             THEN ?STACK
          THEN
       AGAIN ;
```

You may DECOMPILE one word, or a range of words at one time — even the whole FORTH system! This decompliled output may be sent by FPA options to the console, printer, or to disk. DECOMPILE is useful for looking up words, or for obtaining variations of words by decompiling to disk, editing, and recompiling.

SUBROUTINE DECOMPILER The subroutine decompiler finds words called by a specified word to all nesting levels. This makes FORTH PROGRAMMING AIDS especially useful for metacompilers or cross compilers and for finding words called within a loop. The found words may be DECOMPILEd to disk.

CALLFINDER This set of routines is designed to find the calls to a specified word or set of words, either within the context vocabulary or across all vocabularies. Useful to locate and merge infrequently called words, or to see which words will be affected by changing the specified word.

TRANSLATOR This program provides a one-to-one translation of the high level FORTH words in RAM. (This is sometimes called decompilation, but the output is not suitable input for a FORTH compiler). Useful for debugging, patching into compiled words, etc.

System Requirements FORTH nucleus based on the fig-FORTH model or 79-STANDARD; a minimum of 3K bytes and a recommended 13K bytes of free dictionary space.

Ren Curry 415/322-1463 or Tom Wempe 408/378-2811

- -

Yes, send me a copy of FORTH PROGRAMMING AIDS, including all source code and the 40-page manual.

☐ fig-FORTH model	$150	Calif. residents add 6.5% tax.
☐ FORTH-79 STANDARD (specify system)	$150	Foreign air shipments add $15.
☐ Manual alone (credit toward program purchase)	$25	
☐ Send more information		

☐ Master Charge ☐ Visa Account Number _____ Exp. Date _____

Name _____

Company _____

Street _____

City _____ State _____ Zip _____

Indicate disk format:

- ☐ 8" ss/sd fig-FORTH screens
- ☐ 8" ss/sd CP/M™ 2.2 file
- ☐ Apple 3.3
- ☐ PC FORTH
- ☐ Other _____

Send to: **Curry Associates • P.O. Box 11324 • Palo Alto, CA 94306 • 415/322-1463 or 408/378-2811**

A Recursive Decompiler

```
SCR # 55
  0 ( GOES INTO )
  1 : MYSELF LATEST PFA CFA , ; IMMEDIATE
  2    ( REGULAR FIG PFA & LFA )
  3 0 VARIABLE GIN  ( # TO INDENT )
  4 : GIN+ CR GIN @ 2+ DUP GIN ! SPACES ;
  5 : DIN CR GIN @ SPACES ;
  6
  7 : GCHK DUP @ 2+ ' COMPILE =
  8      IF  2+ DUP @ 2+ NFA ID. 2+
  9      ELSE DUP @ 2+ DUP ' LIT =
 10           OVER ' BRANCH  = OR
 11           OVER ' OBRANCH = OR
 12           OVER ' (LOOP)  = OR
 13           SWAP ' (+LOOP) = OR
 14        IF 2+ DUP @ SPACE . 2+
 15        ELSE DUP @ 2+ ' CLIT =
 16         IF 2+ DUP C@ SPACE . 1+
 17         ELSE DUP @ 2+ ' (.") =
 18          IF 2+ DUP COUNT TYPE
 19             DUP C@ 1+ +
 20          ELSE 2+ THEN THEN THEN THEN
 21          -2 GIN +! ;
 22 ( HANDLE SPECIAL CASES )
```

```
SCR # 56
  0 ( GOES INTO )
  1
  2 : (GOESINTO)  ( PFA...)
  3       DUP CFA @ ' ; CFA @ =
  4       OVER ' ERROR = 0= AND
  5  IF ( COLON DEF. & NOT 'ERROR' )
  6    BEGIN DUP @ DUP ' ;S CFA =
  7       OVER ' (;CODE) CFA = OR 0=
  8    WHILE ( HIGH LEVEL & NOT END OF COLON
  9            DEFINITION )
 10       2+ DUP GIN+ NFA ID. KEY DUP 81 =
 11     IF ( 'Q' ) SP! QUIT
 12     ELSE 13 =  ( RETURN )
 13       IF ( GO DOWN ONE LEVEL ) MYSELF
 14       ELSE DROP THEN
 15     THEN GCHK
 16    REPEAT ( SHOW LAST WORD )
 17    2+ DIN NFA ID.
 18  THEN DROP ;
 19
 20 : GOESINTO -FIND IF DROP 0 GIN !
 21   (GOESINTO) ELSE ." NOT FOUND" THEN ;
 22
```

Robert Dudley Ackerman

Editor's Note: A FORTH "decompiler" is a tool that scans through a compiled dictionary entry and tells you what has been compiled. In the case of a colon definition, it prints the names of the words that are pointed to inside the definition. In an ideal programming environment, in which you have the source for your system right on your disk, you may not need a decompiler. But otherwise, it beats all the hit and miss "ticking" and dumping you would have to do. Decompilers can also be useful learning tools.

A very thorough decompiler was written by Ray Duncan of Laboratory Microsystems and published in Doctor Dobbs, September 1981. The following decompiler, while not as complete as Ray's (and not as elegantly written — beware of long definitions), introduces a clever feature: recursive descent. In this version, pressing the space bar steps you through each name used in a colon defintion, but pressing carriage return instead causes the word whose name was just printed to be itself decompiled. This allows you to weave your way through the threaded interpretive code down to any level you want.

On occasion it is desirable to know what words a given word is made up of and what words those words are made up of in turn. Thus the word **GOESINTO**, which naturally calls for recursion. I used **MYSELF** defined with a few standard FIG words.

GIN keeps track of indentation (Goes IN). **DIN** does an indentation (Does INdent). **GCHK** does special cases, particulary where a word is followed by a literal (or a one bite literal, called **CLIT** in Lyon's FORTH). The main word, **(GOESINTO)** is straight-forward. For a colon definition, it goes through each code field printing a name and waiting for a key.

A 'Q' ends execution; a carriage return calls **(GOESINTO)** recursively, printing out the names in the last word shown; any other key continues until a **;S** signals the end of a colon definition, or **(;CODE)** signals a drop into machine language from high level.

One improvement I envision is being able to back up one level, rather than quiting altogether. This would avoid the problem of having to avoid 'error' and other words which use words which use themselves. You could back up one level rather than quiting, not being able to finish the original word. Another improvement would be to use a fence to avoid seeing low level words of no immediate interest.

To use this utility with a Starting FORTH system, change the ticks to bracket-ticks. ' –> **[']** .

Robert Dudley Ackerman is head of the San Francisco Apple Core FORTH Users. □

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.

Technotes

6502'S U/ BUG

Jack Haller
Boonton, NJ

I have discovered a bug in the fig-FORTH 6502 ASSEMBLY SOURCE LISTING RELEASE 1.1 involving the word **U/** . I came across this problem while trying to implement Glen Hayden's "A Serial Day Date Compression which appears in the *1981 FORML Proceedings*. After typing in Mr. Hayden's screens and running them I noticed that after a certain date the program came back with erroneus dates. Upon further debugging I traced the problem to the word **U/** and proceeded to test. I found that the following terminal entries provide a valid illustration of the problem:

1007671. 36525 U/ . 27 OK
(Quotient is correct)
1007672. 36525 U/ . 24 OK
(wrong, should be 27)

Apparently, any unsigned divisor greater than or equal to 8000H will exhibit the symptoms.

```
SCR # 49
  0 ( U/ FIXED                          )   HEX
  1 CODE U/   N 1+  STY,
  2           SEC 2+  LDA,                  SEC     LDY,
  3           SEC 2+  STY,  .A ASL,  SEC    STA,
  4           SEC 3 + LDA,                  SEC 1+  LDY,
  5           SEC 3 + STY,  .A ROL,  SEC 1+ STA,
  6    10 # LDA,  N STA,
  7    BEGIN,    SEC 2+  ROL,    SEC 3 + ROL,    N 1+  ROL,
  8    SEC,      SEC 2+  LDA,    BOT     SBC,    TAY,
  9              SEC 3 + LDA,    BOT 1+  SBC,    PHA,
 10              N   1+  LDA,    0 #     SBC,    PLA,
 11    CS IF,    SEC 2+  STY,    SEC 3 + STA,    THEN,
 12              SEC     ROL,    SEC 1+  ROL,    N DEC, 0= UNTIL,
 13    POP JMP,  END-CODE
 14      DECIMAL     ;S
 15
```

Tracing thru the machine code of **U/** step by step for the two examples above, I found that at the 13th iteration in the division loop the MS word of the dividend assumes the value of 8000H. Since the divisor is 8EADH, the carry to the quotient is not set. The MS word of the dividend is then shifted left, and since the arithmetic is only to 16-bit precision, what should be 10000H becomes 0000H and the next iteration is not valid.

In order to put forth a solution, I have written a modified version of **U/** which uses an extra byte of precision where needed (above). I do not claim that my analysis or solution is the definitive one, but only would like to inform any user about the possible problem.

BACK TO ACKERMANN

Don Russ
Lake Forest, IL

I recently received Volume III, which was most interesting and enjoyable. There was, however, at least one error that may have been brought to your attention by now. On the chance it was not, a working listing of the Fig-FORTH version of the Ackermann function from page 89 follows:

```
( Ackermann function )
: ACKERMANN  [ SMUDGE ]  ( K J – F )
  1 CALLCNT +! OVER
  IF
    DUP
    IF
      OVER OVER 1– ACKERMANN
      ROT ROT DROP 1– SWAP ACKERMANN
    ELSE
      DROP 1– 1 ACKERMANN
    ENDIF
  ELSE
    SWAP DROP 1+
  ENDIF [ SMUDGE ] ;
```

This was transferred to a text file from a FORTH screen that executed with the same results published, so be assured it is typographically correct. May I suggest that the same procedure could make your publication easier to publish and increase the integrity of its contents?

Thanks for your letter. Regarding your final comment, we agree in principle. Unfortunately, the quality of your printer was such that we could not reproduce your listing. Writers, please use black ribbon on plain white paper. —Editor

FLUSH IS TOO MUCH

Bruce Walker
San Pedro, CA

FLUSH in the FIG model of FORTH writes out the memory buffers, and invalidates them at the same time, so that the next time one is needed, it has to be reread from disk again. That is logically fine, but leads to quite a lot of I/O and in the normal edit-compile-test-edit-. . . sequence can be frustrating. The enclosed code writes out the buffers but leaves the buffers still marked as valid. This protects you against disasterous edits or test cases which run away, but still keeps I/O to a minimum. (Curiously, **SAVE-BUFFERS**, the FORTH-79 standard analog of **FLUSH**, is silent on whether the memory buffers are valid after its execution.) I believe that this version is better as **FLUSH** can be defined as

```
: FLUSH FL   EMPTY-BUFFERS ;
```

but **FL** cannot be made out of **FLUSH**.

Naturally, this definition is valid only for Fig-model FORTHs.

```
: FL  FIRST LIMIT FIRST
  – B/BUF 4 + / 0 DO
  DUP @ 0<
  IF DUP @ 32767 AND OVER !
  DUP 2+ OVER @ 0 R/W THEN
  B/BUF 4 + + LOOP DROP ;
```

All true. The current **FLUSH** *combines two logically distinct functions: writing changed buffers to disk, and marking the buffers as empty. The second function, sometimes called a "mount" command, allows you to change disks after a* **FLUSH**. *—Michael Perry*

RANDOM BUGS

Donald P. Madson
Minneapolis, MN

While using the random number generator that has appeared in FORTH Dimensions (III/5, pg. 152), I noticed a problem. If the seed is $3954 the scaling ratio becomes one and the result is equal to the range instead of the range −1. Depending on the program this could cause anything from minor irritations to program self-destruction.

Thanks Don. I'd like to point out that a different high-level random number generator appears on p. 265 of Starting FORTH. The code for this version was provided by Dean Sanderson of FORTH, Inc., and I doubt if there are any problems with it. —Editor

6800 "CONTEMPLATIONS"

Ronald Zech
West Germany

Now that I've worked with FORTH about two years I would like to share some observations regarding the FORTH kernel and especially the 8080- and 6800- listings.

I have implemented the FIG versions of both the 8080- and the 6800-FORTH into several machines, and I think that the 8080 version is fairly bug-free. But the 6800 version isn't! At first, the word **M*** is not included in 6800 FORTH, but copying it from the 8080 version (not in accordance to the FIG model screen 57!) leads to a working completion. But this also requires the inclusion of a "DDUP" word into the 6800 dictionary doing the job of the 8080-fig-word **2DUP** (which is named not following the usual name convention with "Dxxx" for double-length operators; please watch this dangerous practice).

(Editor's Note: The choice of the prefix "2" was not accidental. It was chosen because this class of words handles two 16-bit cells, regardless of whether or not they comprise a double-length number.)

The worst bug in the 6800 FORTH occurs at signed division and is based at the fact that in the word **/MOD** a call of the word **/U** occurs instead of calling the appropriate word **M/** . Furthermore the word **M/** doesn't exist in the 6800 version. (Look also for the words "+− and **D+−** .) When I included **/M** and replaced it with the **U/** in the word **/** then the behavior of the slash-operator was not yet okay.

That lead to the detection of a further bug in 6800- FORTH: the word **S−>D** is defined in the 8080 Listing as a primative similar to screen 56 in the installation manual, but not so in the 6800- definition. Here it is defined in high-level by saving 4 (four!) bytes but missing a conditional branch and its literal. Including this "missing link" lead to a completely exact performance of the signed divide.

But this latter bug leads to a more fundamental question regarding the high-level/low-level balance especially regarding the 6800- version (but with not as great extent also regarding 8080) of FORTH. Especially the 6800 version seems to show the attempt of a rigorous byte-saving. In the special case of the bug above the byte-saving compared to an appropriate primative-definition disappeared completely after debugging the 6800-high-level definition. Compare with the definition below which I'm using for myself instead.

```
      count
      S->, D+80
STOD  (*+2)
      TSX
      LDAB #255
      LDAA 0,X
      BMI  STOD1
      INCB
STOD1 TBA
      JMP PUSHBA
```

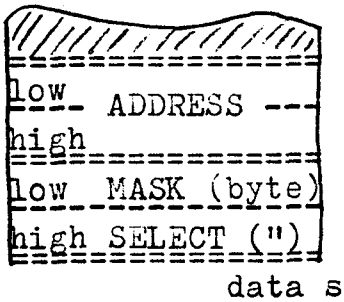
Look also at the word **TOGGLE** in the 6800 listing. I have found this word worthwhile not only as a system word but in aplications. Therefore its time penalty as a high-level word for only toggling a single byte with a pattern is not at all acceptable.

I'll close the 6800-Fig-FORTH contemplations with a little remark regarding the loop run-time procedure: following label XLOOP in that listing you will find the instruction "BRA XPLOP2" which can become replaced by "BRA XPLOF" resulting in a little speed-up of looping and branching with no memory-space penalty.

In some cases it is not time effective and acceptable to make bit-level decisions by using 16-bit-arithmetic. Look for example to hand-shake-procedures at ports in high-speed communication applications. The problem that languages inherently show a time penalty compared to assembler programming leads typically to the smart definition of 'clever' language-elements. Such a smart word I have found in (Charles Moore may forgive me) some Microsoft BASICs: it's the WAIT command.

This is a word which expects an address, a mask and an optional select (the later two are bytes). It fetches the address contents, XOR's it with the select and makes a AND with the mask; if result equals zero then repeat. Select-byte default value is zero. I will suggest the following definition as a exact copy of this construct as follows (6800 code):

```
      count
      'WAI', 'T'+80
      link
WAIT  *+2
      TSX
      LDX  2,X  get addr.
      LDAB 0,X  get (adr)
      TSX
      EORB 0,X  bit polarity
      ANDB 1,x  bit select
      BEQ  WAIT+2
      PULA
      PULA
      PULA
      PULA
      JMP  NEXT
```



I think that it is not good if FORTH isn't able to do a smart thing that BASIC can (although of course FORTH is able to include special assembler-defined words).

NEW PRODUCT ANNOUNCEMENTS

PROGRAMMABLE CONTROLLER WITH SOLID STATE DISK

Controlex Corp.'s CS105, an intelligent controller intended for industrial and process control applications, operates in ROM resident FORTH to allow program development directly on the CS105. System memory is configured as a "solid state disk" to provide fast access and high reliability in hostile environments where rotating memories are failure prone. The system includes RS-232 serial port, cassette interface, printer interface, real-time clock, host/target switch, and protected programming switch for EEPROMS.

The CS105 serves as its own development system in the "host" mode, obviating an emulator and the need for downloading to the target machine. Compiled applications may be saved in the nonvolatile portion of the solid state disk. When the host/target switch is in the target position, the system can easily be configured to boot up the user's application.

Unit price is $2995, including FORTH software and full user documentation.

Controlex Corporation • 16005 Sherman Way • Van Nuys, CA 91406 • (213) 780-8877

FORTH-79 VERSION 2 FOR APPLE AND Z-80 CP/M

FORTH-79 Version 2 for APPLE II/II+, Z-80 CP/M 2.x, and NorthStar DOS users. Floating Point and HIRES Graphics are also available (HIRES on APPLE & NorthStar ADVANTAGE only).

Base system price is $99.95. With enhancements, $139.95 (NorthStar Advantage users add $49.95 to include HIRES).

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

---

pns-FORTH FOR ATARI 400/800

fig-FORTH includes full screen editor, Atari CIO interface and graphics commands, debugging tools, 6502 assembler, string package, tiny multi-tasking kernal, player/missile graphics, sound and manual controller interface.

Requires 32K min., 1 disk drive. 250 page manual covers all features and includes brief FORTH tutorial. $90, includes shipping, manual, diskette and four newsletters. Order from Mountain View Press or directly from:

Pink Noise Studios • P.O. Box 785 • Crockette, CA 94525

SOURCE FOR MARX FORTH

Complete source code for Marx FORTH only $30, sold as an ideas package and tutorial. Includes Z-80 assembler and metacompiler. Features "links first," all math in machine code, 1-byte relative branching, arguments-results, unique compiler security techniques, headerless code, printer control, fast find and 79-Standard. Vendor package available for $450.00.

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

FORTH LANGUAGE CARD FOR APPLE II

Plug this Language Card into Slot 0 and run FORTH without a disk drive. Minimum 16K required; however, it prefers 48K of RAM, because you will then have a 24K byte pseudo disk. The entire pseudo disk can be dumped back to cassette for storage. This Card is compatible with the Apple Integer BASIC Card, with the 8K FORTH dictionary replacing the Integer BASIC. Implemented by Dr. C.H. Ting. Price is $100.00.

OFFETE Enterprises • 1306 S. B Street • San Mateo, CA 94402

DUAL TASKING FORTH

Dual Tasking FORTH by Timin Engineering is the first microcomputer language to permit simultaneous execution of two programs. No interrupts or real time clocks are required, although they may be used if desired. Less than 10% of processor time is devoted to the Dual Tasking function.

Two demonstration programs are included. Requires Z-80 hardware systems with at least 32K RAM and any verison of CP/M or CDOS. $285.

Timin Engineering Company • 6044 Erlanger St. • San Diego, CA 92122 • (714) 455-9008

(Editor's Note: This product is not "the first microcomputer language to permit simultaneous execution of two programs." Several FORTH vendors offer multi-tasking systems.)

---

fig-FORTH FOR INTERACT HOME COMPUTER

Modified for use with cassette. Auto-adjusts to use 16K, 32K, or 48K. Includes FIG line editor, an 8080 assembler and graphics interface.

Only $12 includes cassette and documentation of differences from fig-FORTH.

Russell Schnapp • 8062 Gold Coast Drive • San Diego, CA 92126

---

fig-FORTH ON PET/CBM

fig-FORTH version for CBM disk systems allows up to eight units (16 drives) treated as single mass storage. Employs CBM's screen editor. Also includes FIG editor, 6502 assembler, string package, data-base demo, calendar program, case statement and decompiler. $45 includes two disks and very minimal documentation. Assumes familiarity with fig-FORTH. Include description of your hardware.

Juergen Pfeifer • Oranjerring 28 • 4150 Krefeld • West Germany

Book Review

Discover FORTH: Learning and Programming the FORTH Language

Thom Hogan
Osborne/McGraw-Hill, 1982

Reviewed by Glenn S. Tenney, Fantasia Systems Inc.

In his introduction, the author states that the book will discuss developing work habits that suit the FORTH environment. This goal, however, has not been fully achieved due to many technical errors, omissions and misconceptions.

In describing manipulations of the stack, which is strangely referred to as a "poor man's array," the author unfortunately places the top of the stack to the left. This notation is especially confusing when showing the operation of comparison words.

In the chapter about memory manipulations, the description of **CONSTANT** is evidently based on the misconception that some FORTH implementations do not initialize the constant with a value from the stack when defined.

The chapter on control structures never explicitly states that **LOOP** adds one to the increment. **UNTIL** is described backwards, with loops continuing while true. Booleans are described as "further mathematical possibilities." The chapter concludes with a half page discussion of the virtues of having a **CASE** statement, while never mentioning the word **LEAVE**.

A later chapter states that the variable **BLK** contains the number of the last block accessed.

Discover FORTH concludes with six appendices: a coding sheet, a 79-Standard glossary, a table of ASCII characters, suggested alternatives to the FORTH system, typical error messages and some FORTH extensions. Because the 79-Standard glossary has been rewritten incorrectly in places, that appendix must be ignored.

If all of these errors and misconceptions were corrected, this book could be a good introduction to FORTH. In its current condition, it cannot be recommended. In the meantime, *Starting FORTH* is a much better alternative.

Course Review
Laxen & Harris
By Brian Donovan

Editor's note: The reviewer attended a general FORTH course taught by Laxen & Harris in February, 1982.

This course is fantastic! I went from code I couldn't read myself to implementing multitasking in two days. Kim Harris and Henry Laxen are two incredible programmers and great people to learn from.

FORTH Is much, much more than a language. It is the implementation of an incredibly powerful philosophy for solving complex problems in a way that is not only effective but fun.

Now all this was not apparent to me before this course. In ten sessions, twice weekly, three hours per night, the class went from simple stack operations to metacompilation. More than just an understanding of how to use FORTH's various tools, I learned the difference between "good" FORTH and "bad" FORTH, and why there can be such a difference.

Kim Harris and Henry Laxen have been so intimately involved in the building of what FORTH really is that you can't help but get a feeling for the philosophy, the style, and the community of FORTH.

The Laxen & Harris course is a great way to get up to speed fast. You'd better hurry though; the price of the course limits it to professionals and very serious amateurs.

Upcoming Issues

Here's the planned schedule of themes for the remaining issues of Volume IV, including the deadline for theme articles:

3 Operating Systems	**—**
4 Coding for ROM	**9/01**
5 Business Applications	**10/15**
6 Teaching FORTH	**12/15**

The projected themes for Volume V are: Project Management, FORTH in the Arts, Serial Communications, Laboratory Workstations, The FORTH Environment, and Looking Back (FORTH History).

• The Institute for Applied Forth Research, Inc.

PURPOSE

● Purpose

In the past decade the Forth programming language has emerged as a powerful tool for applying computers. However, we feel a lack of application-oriented information has reduced its general acceptance. Recognizing this, we have chartered a not-for-profit corporation to support and promote Forth and its applications.

PROGRAMS

● Programs

The Institute will sponsor Forth-related conferences such as the 1981 and 1982 conferences at the University of Rochester. Other institutions using Forth may consider hosting conferences or seminars with administrative backing from the Institute. We will address specific topics in Forth through seminars, workshops, and lectures like those held this past May in Rochester.

We are starting a program of summer student fellowships at universities working with Forth, in which the projects will be chosen so as to further a student's expertise in Forth, while doing work of interest to the host institution and others. Further, in recognition of the fact that many students trained in the sciences have not had a chance to apply that science within the constraints of industry, we are arranging summer internships within companies applying Forth. Combination academic/industrial internships are also possible, and would facilitate the transfer of new techniques between the academic and industrial sectors. We welcome suggestions for this program, as well as inquiries from companies and institutions interested in sponsoring students.

Finally, we are establishing a library or archive of Forth-related materials to serve as a resource for the community.

PUBLICATIONS

● Publications

We plan to publish a refereed, professional journal, whose primary subjects will be Forth-based tools and their applications in industry and research. Referees are being chosen, based on their experience and interest, from universities, research laboratories, and businesses using Forth. We intend to publish papers not only by professional Forth programmers, but also by people who have used Forth as a tool to facilitate their own work. The journal will try to represent the growing Forth community, and provide a forum for original work.

The journal will appear twice the first year, and quarterly thereafter. The first issue will come out in January 1983.

The Institute will also undertake publication of the proceedings of the conferences it sponsors. The 1982 Rochester Forth Conference Proceedings should be available in September from Mountain View Press.

● *If you are interested in helping to further the Forth concept through a unique organization, please contact:*

Thea Martin, Executive Director
The Institute for Applied Forth Research, Inc.
70 Elmwood Avenue
Rochester, New York 14611
(716) 235-0168

Fig Chapters

U.S.

Arizona

Phoenix Chapter
Peter Bates at 602/996-8398

California

Los Angeles Chapter
Monthly, 4th Sat., 11 a.m., Allstate Savings, 8800 So. Sepulveda Blvd., L.A. Philip Wasson 213/649-1428

Northern California Chapter
Monthly, 4th Sat., 1 p.m., FORML Workshop at 10 a.m. Palo Alto area. Contact FIG Hotline 415/962-8653

Orange County Chapter
Monthly, 3rd Sat., 12 noon, Fullerton Savings, 18020 Brockhorst, Fountain Valley. 714/896-2016

San Diego Chapter
Weekly, Thurs., 12 noon. Call Guy Kelly, 714/268-3100 x4784

Massachusetts

Boston Chapter
Monthly, 1st Wed., 7 p.m. Mitre Corp. Cafeteria, Bedford, MA. Bob Demrow, 617/688-5661 x198

Michigan

Detroit Chapter
Call Dean Vieau, 313/493-5105

Minnesota

MNFIG Chapter
Monthly, 1st Mon. Call Mark Abbot (days) 612/854-8776 or Fred Olson, 612/588-9532, or write to: MNFIG, 1156 Lincoln Ave., St. Paul, MN 55105

New Jersey

New Jersey Chapter
Call George Lyons, 201/451-2905 eves.

New York

New York Chapter
Call Tom Jung, 212/746-4062

Oklahoma

Tulsa Chapter
Monthly, 3rd Tues., 7:30 p.m., The Computer Store, 4343 So. Peoria, Tulsa, OK. Call Bob Giles, 918/599-9304 or Art Gorski, 918/743-0113

Oregon

Portland Chapter
New Chapter! Call Timothy Huang, 9529 Northeast Gertz Circle, Portland, OR 97211, 503/289-9135

Pennsylvania

Philadelphia Chapter
New Chapter! Call Barry Greebel, Continental Data Systems, 1 Bala Plaza, Suite 212, Bala Cynwid, PA 19004

Texas

Austin Chapter
Call John Hastings, 512/327-5864

Dallas/Ft. Worth Chapter
Monthly, 4th Thurs. 7 p.m., Software Automation, 1005 Business Parkway, Richardson, TX. Call Marvin Elder, 214/231-9142 or Bill Drissel, 214/264-9680

Utah

Salt Lake City Chapter
Call Bill Haygood, 801/942-8000

Vermont

ACE Fig Chapter
New Chapter! Monthly, 4th Thur., 7:30 p.m., The Isley Library, 3rd Floor Meeting Rm., Main St., Middlebury, VT 05753. Contact Hal Clark, RD #1 Box 810, Starksboro, VT 05487, 802/877-2911 days; 802/453-4442 eves.

Virginia

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

Washington

Seattle Chapter
Call Chuck Pliske or Dwight Vandenburg, 206/542-7611

FOREIGN

Australia

Australia Chapter
Contact Lance Collins, 65 Martin Rd., Glen Iris, Victoria 3146, or phone (03) 292600

Canada

Southern Ontario Chapter
Contact Dr. N. Solnseff, Unit for Computer Science, McMaster University, Hamilton, Ontario L8S 4K1, 416/525-9140 x2065

Quebec Chapter
Call Gilles Paillard, 418/871-1960 or 643-2561

England

English Chapter
Write to FORTH Interest Group, 38 Worsley Rd., Frimley, Camberley, Surrey, GU16 5AU, England

Japan

Japanese Chapter
Contact Masa Tasaki, Baba-Bldg. 8F, 3-23-8 Nishi-Shimbashi, Minato-ku, Tokyo, 105 Japan

Netherlands

HCC-FORTH Interest Group Chapter
Contact F.J. Meijer, Digicos, Aart V.D. Neerweg 31, Ouderkerk A.D. Amstel, The Netherlands

West Germany

West German Chapter
Contact Wolf Gervert, Roter Hahn 29, D-2 Hamburg 72, West Germany, (040) 644-3985

SPECIAL GROUPS

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

Nova Group Chapter
Contact Mr. Francis Saint, 2218 Lulu, Witchita, KS 67211, 316/261-6280 (days)

MMSFORTH Users Chapter
Monthly, 3rd Wed., 7 p.m., Cochituate, MA. Dick Miller, 617/653-6136

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

0582 S
R L SMITH
ESL SUBSIDIAR OF TRW
PO BOX 510
SUNNYVALE, CA 94086

Address Correction Requested