Volume 6, Number 1

FORTH Dimensions

May/June 1984
$2.50

fig-Forth Interpreters

New Control Structure

Anonymous Variables

Interactive Editing

Using Apple IIe's Extra RAM

FORTH Dimensions
Published by the Forth Interest Group
Volume VI, Number 1
May/June 1984

Editor
Marlin Ouverson

Production
Jane A. McKean, Et Al.

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

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

Symbol Table

Simple; introductory tutorials and simple applications of Forth.

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

Advanced; requiring study and a thorough understanding of Forth.

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

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

fig — Code and examples conform to fig-FORTH.

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

FORTH Dimensions

FEATURES

DEPARTMENTS

Letters to the Editor

NOT and LEAVE Background

Dear Marlin,

I can see from reading the letters to the editor in recent *Forth Dimensions* that there is a lot of interest and programming being done in the new Forth-83 Standard. I was especially interested in Leo Brodie's letter about **NOT** and **LEAVE** and have some background information on the development of these ideas.

In the 1979 Standard the other typical logical operations were available: **AND XOR OR** and the two's complement; yet for unknown historical reasons the one's complement was absent. When using a one's complement, one was left with second-best alternatives such as **-1 XOR** (six bytes on a fig-FORTH system) or **NEGATE 1-** (four bytes with Forth-79). Also, the naming issue was unclear; the name given in the Uncontrolled Reference Word Set of Forth-79 was **COM**, I had coded **INV**, and other possible names included **COMP**, **COMPLEMENT** and **INVERT**.

At least as early as 1981, **NOT** was proposed as a one's complement for the Forth standard ("The Nature of the Forth Standard," Hans Nieuwenhuyzen, 1981 Rochester Forth Standards Conference, p. 91; "Some Thoughts on the Forth-79 Standard," Rieks Joosten, 1981 Rochester Forth Standards Conference, pp. 134-5; "**NOT** vs. **COMP**," Hans Nieuwenhuyzen, 1981 Rochester Forth Standards Conference, p. 149; "Some Concepts in Forth," Rieks Joosten, 1981 Rochester Forth Standards Conference, pp. 328-9). But this was when a flag output was one.

When Forth-83 standardized **NOT** as the one's complement, it was a simplification allowed by the new flag, in which all bits are set to one. The common usage, such as **0< NOT**, was not disturbed. A redundancy in the 1979 standard, **0=** and **NOT**, was eliminated. The hole left by the missing one's complement was plugged. Furthermore, the naming issue was eliminated without controversy. All this was accomplished with no additional cost in new words to the standard. This result was better than anything I had seen reason for which to hope.

This gain was not without some cost. An example of a common phrase which required re-thinking was **-TEXT NOT**. This, of course, is most simply re-phrased **-TEXT 0=**. **NOT** could no longer be used with any number before an **IF**; the input to the **NOT** had to be a pure flag. However, full functionality was maintained as **0=** took over where **NOT** didn't operate in the same manner.

I also have some information on the history of **LEAVE**. The old **LEAVE** technically set a flag; it was a programming trick that functionally encoded the leave flag into the index and limit. After executing **LEAVE** the loop body would continue to execute until encountering the **LOOP** or **+LOOP**. What I recall of the old **LEAVE** was the extra massaging required to allow the loop to execute part of an iteration after it was done. The old **LEAVE** was like the new **LEAVE** in that it could be used any number of times within a do-loop at any nesting level of other control structures.

The earliest I had heard of the new **LEAVE** was in conversation with Robert Patten in August 1981. The idea was in print the following November ("A Generalized Forth Looping Structure," Robert Berkey, *3rd FORML Conference Proceedings*, November 1981 republished in *1981 FORML Proceedings*, Vol. One, pp. 31-7). Attracted by the do-loop structure in that paper, Robert L. Smith wrote "An additional difference from previous **DO LOOP**s is that **LEAVE** will cause the loop to be exited at the point that it is executed. In my opinion that is an improvement..." ("An Experimental Proposal for **DO**, **+LOOP** and **LEAVE**," Robert L. Smith, Forth Standards Team meeting, May 1982, Proposal 83).

The following articles focused on **LEAVE**: "The existing **LEAVE** usage can work with the new **DO...LOOP**." ("**LOOP&-LEAVE**," Klaxon Suralis, FIG-Tree modem conference, July 21, 1982; republished as "Forth-79 Compatible **LEAVE** for Forth-83 **DO...LOOPS**," *Forth Dimensions* IV/3). "**LEAVE** should **NOT** Jump" (Definition of **LEAVE**," Forth Standards Team meeting, October 1982, Proposal 231). "A jumping **LEAVE**...is incompatible with Forth-79" ("Non-**IMMEDIATE** Looping Words," Klaxon Suralis, 4th FORML Conference, October 1982). But the idea that prevailed was, "A widely desired change has been to branch directly from **LEAVE** to the continuation" ("Leavable Do-Loops: A Return Stack Approach," George B. Lyons, 4th FORML Conference, October 1982).

The history of **LEAVE** is tied in with the history of the do-loop. The 1979 standard found the do-loop to be so controversial that the standard itself said that further consideration was likely ("Forth-79," Forth Standards Team, first edition p. 16; second edition, p. 17). Still today this loop is routinely misused to scan addresses which may fall on the 32K addresss boundary (see, for example, *The Journal of Forth Application and Research*, December 1983, p. 7).

Even before the 1979 standard went into print (October 1980) the existence of the 64K circular loop was known. In early 1980, today's loop idea was in prebirth form as a bug in the routine **32760 10 TYPE**. By July of 1980 the bug was fixed and the 64K circular loop idea had been discussed with Bill Ragsdale and Guy Kelly; it was pubished at the 1981 Asilomar conference (Berkey, *op. cit.*). Robert L. Smith recast the ideas into a conventional format and during early 1982 promoted the structure at Northern California FIG meetings and in print ("Forth Standards Corner," *Forth Dimensions* III/6). At the Washington standards team meeting in May 1982, Andy Wright reported that he had been using a 64K circular loop since 1978 in his own programming language, a language ancestrally aligned with Forth. But the new loop did not allow the old **LEAVE** implementation.

Going back to Leo Brodie's letter, his discovery ... **IF DROP LEAVE THEN +LOOP** was published in the original paper covering the 64K circular loop (Berkey, *op. cit.*, pp. 4, 35.). A variety of implementors beyond those listed above, incuding parties outside the standards team, studied and implemented these various ideas during the course of the standardization process. Before the standard was approved (June 1983), essentially complete Forth-83 systems were running substantial applications.

Any standards group, no matter what the field of endeavor, considers the choices available and picks what is deemed best. Almost by definition not everyone is fully satisfied. When the 64K circular loop made the old **LEAVE** trick unworkable, the alternative deemed best, today's **LEAVE**, was selected.

The standard as a whole has now been implemented by many authors including commercial vendors without abandoning the requirements of the standard, and major applications are running on these systems. Today, as thorough readers of *Forth Dimensions* are aware, there are a variety of implementations, both commercial and public-domain systems, encompassing the complete Forth-83.

The chairman of the standards team has announced that proposals for changes to the standard will not be acted upon before the 1986-1987 time frame. I have my own list of changes I'd like to see made, but the conclusion to be drawn is that the Forth-83 Standard is technically solid, useful and working, and isn't going to change for a long time.

I was impressed during the 1982 standards team meetings by the general unanimity of thought and action which prevailed. The low level of criticism which has followed is itself a measure of the general acceptance and support for the standard. Coming from a community rooted in individualism and non-conformism, the Forth-83 Standard is a significant achievement brought about by the work and compromise of many.

Sincerely,
Robert Berkey
2334 Dumbarton Ave.
Palo Alto, California 94303

Standard Support

Editor:

Leo Brodie's description of traps to watch out for in converting programs with **NOT** or **LEAVE** to the new Forth-83 Standard will help others who are converting similar programs. The experiences described do not indicate any weakness in the standard, rather one-time adjustments to changes which were made for good reasons.

On **NOT**, some earlier Forths had two words — **NOT** and **0=** — for the identical function. Forth-83 avoided this redundancy and made **NOT** the proper bit-wise operator to be used with **AND**, **OR**, etc. The new standard left **0=** unchanged.

LEAVE was changed for several reasons, concerning both speed and generality.

No standard can meet all needs and desires. Most Forth software products will have good reasons to use a few non-standard words and are expected to do so. Few, if any, would object to that practice, provided that the non-standard routines are documented if source code is distributed.

The central purpose of Forth-83 is to overcome needless communication and incompatibility problems caused by the existence of dialects which developed historically but no longer serve any purpose. This new standard is not perfect, of course, but it is an excellent basis for building on the agreements we can achieve and for moving beyond the confusion of dialects. I support it in my work, and urge others to do so.

Sincerely,

John S. James
Member, Forth Standards Team
P.O. Box 1807
Los Gatos, California 95031

More on WITHIN

Dear FIG:

After reading "Within WITHIN" in *Forth Dimensions* (V/5), I thought Mr. Nemeth might be interested in a version of **WITHIN** I have been using for some time:

```
: WITHIN ( lower, upper, n--boolean )
DUP >R MIN MAX R> = ;
```

My **WITHIN** returns a true if n is logically within the upper and lower limits, inclusive. This means that it tends to produce non-meaningful results if the operands fall outside the range of sixteen-bit two's complement numbers, at least with my implementation of **MIN** and **MAX**. Its redeeming value, however, is that it is easily re-written to expect different stack structures and can therefore be used with a minimum of operation overhead. To accept the upper limit before the lower, swap **MIN** and **MAX**. **SWAP** or **ROT** can be tacked to the beginning of the definition, or used before it, to allow limits to be placed anywhere with respect to the number to be checked.

Rich Leggit
P.O. Box 6607
Salinas, California 93912

Obstructing Knowledge?

Editor:

I would like to add my comments to those of Mr. William A. Paine, which were published in the letters section of the September-October issue.

It occurs to me that too much of the literature is geared to those who might be termed "computer freaks" (people who know the inside workings and all the technical aspects of computers) and too little is geared towards people like myself -- people who want to program to meet specific personal needs and who don't feel that it is necessry to be adept in seventeen languages and have degrees in calculus and electrical engineering in order to be decent programmers.

I, like Mr. Paine, would like to see in-depth evaluations of various Forth systems that are available, and I would like comments on how suitable these versions are for beginners as well as those intimately familiar with the inner secrets of silicon chips.
Thanks very much.

Chuck Larrieu
P.O. Box 294
Corte Madera, California 94925

Homebrew TI

Dear Sir:

I have a homebrew TI 9995 computer with fig-FORTH in an 8K EPROM. I

Editorial

A Few Changes . . .

FIG grows as public interest in Forth expands: more chapters and more members help us to expand our services and remind us to keep in mind new members and the public when planning our activities. *Forth Dimensions* is keeping pace with the growth by utilizing new design elements which we hope will equal the high quality of our contributors' work.

Most noticeably, of course, our cover has changed. By putting the table of contents on the inside we can provide more information about each of the items it contains. A short abstract helps find articles of interest and, as time goes on, will be of more help than the title alone when searching for needed reference material.

You will notice that each of our regular departments now has its own logo. This has been done in order to provide visual distinction from other articles when thumbing through the pages. And each article is now keyed to the general degree of difficulty of the material covered. The "thermometer" indicators should prove most useful to programmers new to Forth's concepts. Easy-to-understand applications and introductory tutorials are termed simple; larger applications and explanations of advanced concepts are labelled intermediate; and difficult material is shown as advanced. Writers will want to keep these categories in mind when submitting their work for publication.

Henry Laxen, Forth's programming techniques pro, has taken a couple of months off while teaching for the University of California. His column will be welcomed back in the next issue. Keep your mental faculties finely tuned, as Henry has planned some topics of special interest!

Finally, we would like to welcome the Forth Interest Group's new Board of Directors. Sitting on the new board are John D. Hall, Kim Harris (incumbent), Thea Martin, Robert Reiling and Martin Tracy. A deep debt of gratitude is owed to outgoing board members Bill Ragsdale, Dave Boulton, John James and Dave Kilbridge, all of whom have served from the day of FIG's inception. Their contributions have shaped FIG's history, and we enjoy their continued participation and support. Our special thanks goes out to each of them.

-Marlin Ouverson
Editor

---

was an assembly language bug until you converted me. Still, I like to relapse occasionally and since I have an excellent sixteen-bit assembler in my monitor, I wanted to use it for my Forth assembler.

My best solution to date is simply to define a defining word **MACRO**:

(machine address) **MACRO** (word)

such that when word is executed the machine language program is run. The program is created by the Monitor Assembler. Maybe there is a better way to use an existing assembler in Forth programming. A sixteen-bit assembler is quite complex. The TI 9995 has many illegal codes which cause an interrupt and could also be used for machine language definitions.

Incidentally, in your 9900 listing — which is excellently documented — I found that a warm start at line 0146 needed a value of 11A2 in order to work. Maybe your 99/4A users have found this.

Forth is great fun and I hope to use it extensively for musical and graphical applications.

Yours truly,

R.J. Mitchell
Star Route
Spencertown, New York 12165

Keeping Time

Dear Editor:

In the very interesting article "Timekeeping in Forth" by Bill Ragsdale (*Forth Dimensions*, V/5) there occurs this quote: "Some contemporary applications of keeping time use 0000 hours for midnight."

This sounds as if the author believes the 0000 to be the deviation, when it is indeed the standard. Midnight is described with 0000 hours. The only time 2400 is used is if an event ends exactly at midnight (ref. ISO-3307 standard).

As an aside, the twenty-four hour clock is so much superior over the traditional twelve-hour clock that I am surprised it is not more widely used in this country. For one thing, it eliminates the confusion of what 12:00 p.m. is; not everyone will believe that it is noon.

Perr Cardestam
P.O. Box 32572
San Jose, California 95152

Dear FIG,

I believe you do your readers and Bill Ragsdale a great disservice by requiring a magnifying glass to read the screens in his article "Timekeeping in Forth."

Continued on page 31

President's Letter

New FIG Directors

From the Desk of Bill Ragsdale

Many exciting activities are presently happening for the improvement of FIG, and I'd like to take this opportunity to bring all of the membership up to date. According to the FIG by-laws, membership is organized into professional members (everyone) and voting members (directors). The directors are elected for three-year terms. Our long-term desire is to expand the representation to the full membership as our organizational structure develops.

In previous years, the directors' election was a modest formality, as candidates other than the founding directors were conspicuous by their absence. In other words, we carried on by momentum.

On April fifth of this year, the directors of the Forth Interest Group held their annual election meeting. This year's meeting proved to be a breath of fresh air. Two months earlier, a nominations committee was appointed consisting of Larry Forsley, Marlin Ouverson, Ray Duncan and Gary Feierbach. By a round-robin telephonic process they developed a slate of candidates. At our March business meeting, additional nominations were made from the floor. We asked that candidates confirm their interest by either attending the election or by submitting a short statement of their desires for the future of FIG.

During the April fifth meeting, attended by about seven observers and candidates, the voting members elected the new directors. The tenor of the election was to have a wide geographical and interest representation reflected in the new directors. Elected were Bob Reiling, John Hall, Kim Harris, Thea Martin and Martin Tracy. This new group will represent your interests in policy formation and the selection of FIG's operating officers.

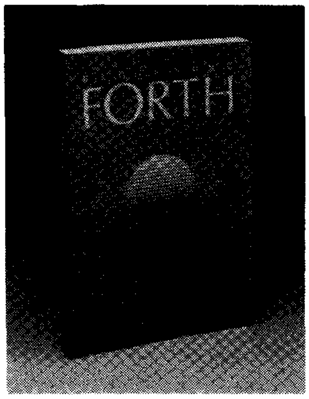
On behalf of the membership, I would like to offer hearty thanks to the outgoing directors: Dave Boulton, Dave Kilbridge, Kim Harris (re-elected), John James and myself. These original founders have guided FIG for the last seven years.

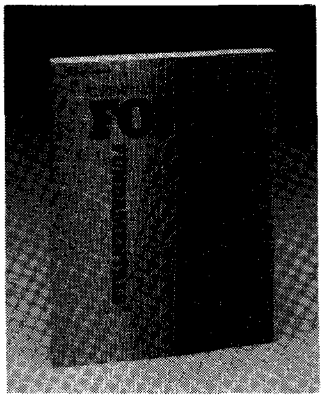
By the time you read this, the officers will have been selected by the directors for the coming year. We expect to expand the breadth of our membership service through the work and expertise of the new officers.

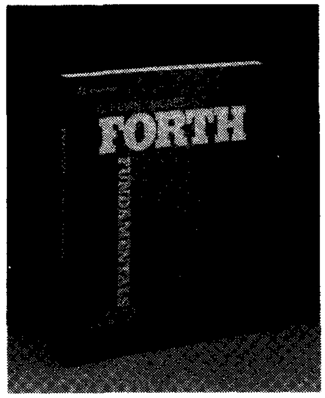
In the next issue, you'll hear more of our welcome of Shepherd Associates, recently appointed as FIG's management firm. I'd like to wrap up this month's letter by expressing my gratitude, and indirectly that of the membership, to the staff of Martens and Associates. Roy Martens, Sari Martens and Betty Mattox have answered the FIG hotline, responded to your correspondence, published *Forth Dimensions* and handled all mail orders for the last three-and-a-half years. They have done the thousand-and-one tasks that facilitated our membership growth from 2500 (in 1980) to the present 4713 members. Only their successful growth as Mountain View Press has necessitated our transition to an outside, independent management firm. Thanks, Roy!

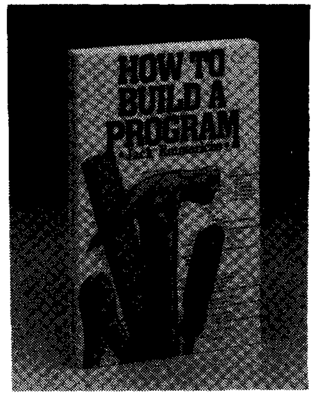
My gratitude is also offered to all of you who have made my five years as President of the Forth Interest Group such an exciting and informative period of my life.

–Bill Ragsdale


FORTH




FORTH


HOW TO BUILD A PROGRAM

Ask the Doctor

Moving to ROM

William F. Ragsdale
Hayward, California

"Ask the Doctor" is Forth Dimensions' *health maintenance organization for queries, requests, help in locating suppliers, applications and aid in understanding the subtleties of Forth. When needed, our columnist will call in specialists in the peculiarities of vendors' hardware variations affecting Forth operation.*

As he begins his second column, we find the doctor in his surgical blues, scrubbed and using his scalpel to deftly excise the first of your letters from its envelope.

Steve Armstrong of Milwaukee, Wisconsin asks, "I have a copy of fig-FORTH 1.4 for the Atari 800. I wiped out the documentation screens starting at screen #40. How do I edit? Has there been an update to this version? Is there a glossary published?"

Rx: fig-FORTH is published in paper listing form by the Forth Interest Group. Vendors, users groups and some computer manufacturers have then customized the listing for their customer/member use, still identifying it as fig-FORTH. The symptoms you describe are incomplete; who is the implementor or distributor of your version? That source is the best starting point for the specifics of your editor or documentation.

However, there are other sources of help. First, the FIG chapters listed periodically in *Forth Dimensions* can connect you with others in your local area. Next, your Atari users group should be familiar with library fig-FORTH versions (such as the Coin-op Division version or the Atari Program Exchange version). Finally, you may unknowingly have a copy of a commercial product. In that case, you should consider purchasing the product with documentation and support.

There have been no updates published by FIG in the last four years. We hope the vendors or supporting user libraries will keep their running system up to date. For example, the Apple Corps in San Francisco distributes a version developed from fig-FORTH by George Lyons. It has extensive documentation and they offer aid from their knowledge base on the Apple II.

The glossary common to all fig-FORTHs is contained in the *Installation Manual and Model* (authored by the good doctor). Consult the back cover of this issue for ordering information.

J. Read of West Beach, Australia laments, "I am the frustrated owner of a BBC micro, Model B with an eighty-track disk drive. The Acornsoft version of Forth will not run even while using tape only. I suspect trouble with the presence of the DFS chip."

The good doctor is stumped by this one. We've seen none of the machines west of Land's End in Cornwall. This is another case of acute hardware dependency. I've taken the liberty of forwarding your letter to Lance Collins of our Melbourne Chapter.

Edward Avila entreats, "Do you have advice on where I may get further information on converting a RAM-based Forth to operate from ROM? I am planning to build a ham radio repeater control system using a one-board computer."

Rx: This question is often asked. As products are transplanted from an interactive environment to a dedicated product, significant changes are needed in technique. This is variously called "target compilation," "meta-compilation" or "cross-compilation."

The general method is to use a Forth program (target compiler) to translate your source program to the Forth object code form, placing the result on disk. This object code is accompanied by machine code for word definitions used by your application. The complete application is then copied to ROM and executed in the final product. Often there is no terminal or disk storage. Target-compiled applications generally range from 1K to 20K in size.

At present, there are no complete books on this process. John Cassady's MetaForth ($30 from Mountain View Press gives three compilers but they only compile to RAM, and John gives only four pages of text and theory. Two other authors are reported to be preparing comprehensive texts. Jerry Boutelle (author of the Nautilus cross-compiler) provides a discussion of the process, with an example, in the *Proceedings of the 1980 FORML Conference*, pp. 111-121.

The generation of programs to run in ROM is complicated by two elements. First, it is usually desired to conserve memory space by removing unneeded word definitions and all word headers. This makes testing quite a bit more difficult than if all of Forth is present. Second, words specifying RAM must be re-designed to operate with separately allocated read/write memory rather than using memory within the program (which is ROM).

Mastery of target compilation also defines a significant career opportunity. Your question illustrates a need area in which educators and vendors could apply their talents. The most direct educational path to your goal would be to take a Forth Inc. (Hermosa Beach, California; 213-372-8493) course on target compilation. The cost is about $900 and requires one week in residence. Inner Access Corp. (Belmont, California; 415-591-8295) occasionally teaches an advanced class which touches on the topic. Nautilus Systems (Santa Cruz, California, 408-475-7461) sells a target compilation system for several processors for $250. It is also sold by Mountain View Press and Laboratory Microsystems.

Lastly, Henry Laxen wrote a series of articles on this topic for *Forth Dimensions* (IV/6, V/2 and V/3). The material is advanced, but the series serves as a good introduction to the above-mentioned products.

George Jones of Lower Hutt, New Zealand writes, "Do you know where we can buy a fig-FORTH listing for the Intel 8096 micro-controller? Is anyone developing one? What help is available for program development for one-chip processors?"

Rx: fig-FORTH implementation pretty much ended in 1980. Few people appear willing to commit effort to a dialect that has been supplanted by Forth-79 and, presently, Forth-83.

Some vendor work is available for one-chippers. Forth Inc. has some target compilers that may be of use. Elizabeth Rather reports that they have one for the 8048 and were working with Intel on an offering for the 8096. Intel dropped the project, although they mentioned it in some sales literature. Forth Inc. would be pleased to offer support in this area, but on a project basis rather than as a standard product.

This issue of target compilation and cross development is a lively issue, as mentioned in the answer above. (Translate "lively issue" to "commercial opportunity.")

Rockwell offers the R65F11 one-chip processor which forms its own development system and EEPROM blower in seven chips on a four-inch square board! This is an enhanced 6502 fig-FORTH in ROM, two timers, an ASCII serial port and many I/O ports. Rockwell's documentation is excellent. Randy Dumse at New Micros, Inc. (Grand Prarie, Texas; 214-642-5494) sells the board wired and tested or in kit form. Randy's documentation is still under construction and leaves much to be desired.

Again, Mr. Jones' question points out to software vendors and application practitioners that sufficient demand exists to support the ROM-based application needs.

Bill Carlson of Cortland, New York asks if the doctor could supply information on the FIG model. "I understand that when the fig-FORTH model was first released, the editor string-comparison word **MATCH** was supplied in code, but the present version gives it in high level. As I have a 5602 I'd appreciate the original form. How can I get a copy? A stamped envelope is enclosed."

Rx: Until August of 1980 the *fig-FORTH Installation Manual* had a machine code version of **MATCH**. In response to repeated member complaints on portability from non-6502 users, I adapted a version by Peter Midnight written in high-level form. It was assumed that those wanting a higher speed would re-code for their processor. The high-level form first appeared in printings marked November 1980. Bill Carlson's envelope has been returned containing a prescription for the requested **MATCH**.

William F. Ragsdale was the founding President of the Forth Interest Group. Bill has authored articles on Forth and its use for BYTE, Forth Dimensions *and* Dr. Dobb's Journal. *As the author of* fig-FORTH Installation Manual and Model, *his work has been translated to run on eleven processors. His memberships include the Forth Standards Team, Society of American Magicians, IEEE and ACM. Bill is the President of an electronics manufacturer and a graduate of the Univeristy of California at Berkeley in Electronics Engineering.*

fig-Forth Interpreters

C. H. Ting
San Mateo, California

"Examining a leopard through a pipe, you can see only one spot."
—An old Chinese proverb

"What is an outer interpreter and what is an inner interpreter?" This is a question often asked by Forth enthusiasts. It is not very easy to answer because it involves the very essence of Forth as an operating system and as a programming language. If we can answer this question satisfactorily, we should be able to cut through much of the mythical fog often surrounding Forth. Examining Forth as an interpretive language is one way of looking at this strange beast. It may not bring instant understanding of Forth, at least it will put you several steps further ahead in appreciating the mechanism which makes it tick.

I did a little research in Forth literature, looking for references on interpreters. The consensus is that there are two interpreters in Forth: a text (outer) interpreter and an address (inner) interpreter. The best verbalization of these concepts is that by Linda Baker and Mitch Derick[1]:

> The outer interpreter is a text interpreter (like a BASIC text interpreter). It parses text from the input stream and looks each word up in the dictionary. When a word is found in the dictionary, it is executed by calling the inner interpreter.
>
> The inner interpreter is an address interpreter (like Pascal's p-code interpreter), which executes definitions whose absolute addresses have been previously compiled into the dictionary.
>
> The two-level mode of interpretation gives Forth both compactness and high speed of operation.

The text interpreter reads the input text stream and translates the text commands to execution addresses, which are turned over to the address interpreter for execution. Most of the compiled commands in the dictionary are represented by lists of execution addresses which, again, can be executed by the address interpreter. In this sense, it is quite all right to equate the address interpreter to the inner interpreter. By execution address is meant the code field address of a dictionary entry. The address interpreter causes the CPU to jump to the address contained in the code field, i. e., an indirect jump through the code field. This indirect jump is the reason why Forth code is called indirect threaded code.

Text Interpreter

The text interpreter is the heart of a Forth system. In the fig-FORTH Model, the dictionary can be roughly divided into three sections: the nucleus, which consists of code definitions; the Level I words, which build up the text interpreter; and the Level II words which are enhancements and utilities over the text interpreter. If we were to pick one word to represent the text interpreter, no doubt it would be **INTERPRET**, which performs the parsing of the input text stream, dictionary searches, number conversion, invoking the inner interpreter, and even compilation of colon definitions.

INTERPRET is a beautiful piece of code, a classic example of the simplicity and power of the Forth language in describing complicated computational processes using high-level words. It is worth the time required to read the code and to do our best to gain the fullest understanding of it. The definition of **INTERPRET** reads as in figure one.

-FIND is a very big word. It first parses a word out of the input stream and places it in the word buffer on the top of the dictionary. It then searches through the dictionary for a command with the same name. If a command is found, its parameter field address is placed on the data stack followed by a true flag. Then **STATE** is examined. If **STATE** is zero, the parameter field address is converted to the code field address, which is then turned over to **EXECUTE**. **EXECUTE** executes this command by invoking the appropriate inner interpreter, which we shall discuss in a moment.

If **STATE** is non-zero, indicating that we are in the compiling state, the parameter field address is converted to code field address and compiled to the top of the dictionary by , (comma). In this fashion, the text interpreter is dubbed as the compiler of high-level colon definitions. Charles Moore took advantage of the great similarity between the text interpreter and the colon-definition compiler and rolled them into a single piece of code. After the command is located in the dictionary and the code field address is available, the interpreter executes it with **EXECUTE** or the compiler compiles with , .

Now, if **-FIND** failed to find a command with a matching name, control is passed to **NUMBER** which converts the parsed word to a double-precision number on the data stack. If a period was embedded

```
: INTERPRET ( Interpret or compile source text input words )
  BEGIN - FIND ( Parse out a word and search dictionary )
    IF ( Found ) STATE @ <
      IF CFA, ELSE CFA EXECUTE THEN
    ELSE HERE NUMBER DPL @ 1+
      IF [COMPILE] DLITERAL
      ELSE DROP [COMPILE] LITERAL THEN
    THEN
    ?STACK
  AGAIN ;
```

Figure One
The definition of INTERPRET

in the number string, which causes **DPL** to differ from -1, the double-precision number will be processed by **DLITERAL**; otherwise, the high-order part of the double number is dropped from the stack and the remaining single-precision sixteen-bit number will be processed by **LITERAL**. If you look over the definitions of **DLITERAL** and **LITERAL**, you will find that they also examine first the contents of **STATE**. If **STATE** is zero, indicating an executing state, the number is left on the data stack. If the **STATE** is non-zero, indicating a compiling state, the number will then be compiled on top of the dictionary, either as a double-precision literal or as a single-precision literal.

After any of these four paths is taken, the data stack is checked for underflow by **?STACK**. If the stack is okay, control returns to the beginning of the loop to process the next word in the input stream. **INTERPRET** is an infinite loop without an explicit exit point. This loop may be terminated by three conditions: **NUMBER** failing to convert a word into a valid number, **?STACK** detecting stack overflow or underflow, or reaching the end of input stream.

The actions taken by the text interpreter can also be described by a conventional flowchart, as shown in figure two. It clearly illustrates the four alternative paths after a word is parsed out of the input stream: a command is executed, a code field address is compiled, a number is left on the data stack, or a literal is compiled.

The text interpreter in Forth is simple because it only has to deal with two types of information: names of commands in the Forth dictionary and numbers. It does not have to know anything about the commands other than their names. From a name, the text interpreter can find the code field address of the corresponding command, and uses this address for execution or compilation. All these things can be done in one pass, without complications that have to be dealt with in other high-level languages.

Inner Interpreters

According to the definition of the inner interpreter we mentioned before, one could take the word **EXECUTE** as the inner interpreter. But equating the inner interpreter to the address interpreter really does not bring the characteristics of Forth into sharp focus. To fully appreciate the power of Forth and to understand its inner machinery, we have to dig one level beyond this indirect jump and investigate what happens after the jump. I would like to call these routines inner interpreters, to which execution control is steered by the code fields. These routines actually determine what the particular Forth word does and how the information stored in the parameter field is to be processed or "interpreted."

To restrict the inner interpreter to mean only the address interpreter leaves us with only a partial understanding of a very fundamental characteristic of Forth. Similar to other high-level languages, Forth has many different classes of commands. However, instead of burdening the text interpreter with the very complicated task of classifying them, Charles Moore chose to use many interpreters, each tailored to a class of commands. By factoring the syntax analysis out of the text interpreter and dealing with different classes of commands by steering them to appropriate inner interpreters via the code field, he preserved the simplicity of the text interpreter and also enhanced its ability to handle a variety of commands and data structures.

Let me first propose a formal definition of inner interpreters and then elaborate with more detailed discussion and examples.

> Inner Interpreters
>
> The set of execution procedures, usually in the machine code of the host computer, which execute various Forth words by processing the information stored in their parameter fields. The address of such a procedure is stored in the code field of a Forth definition. Forth definitions of the same class have the same address in their code fields.

Following this definition, we can identify several inner interpreters in a Forth system. Since the inner interpreters are not regular Forth definitions, they were not defined in 79-Standard or 83-Standard. I have to pick their names from the fig-FORTH Model. Here is a list of the inner interpreters used in fig-FORTH:

DOCOL Address Interpreter
DOCON Constant Interpreter

DOVAR	Variable Interpreter
DOUSE	User Variable Interpreter
DOVOL	Vocabulary Interpreter
.+2	Code Interpreter

In a code definition, the address stored in the code field is the parameter field address. In a sense, the machine-code routine in the parameter field is the inner interpreter of this code definition. However, it is much more logical to group all the code definitions in one class and let **.+2** serve as the inner interpreter for the whole class.

Create New Inner Interpreters

The above list of inner interpreters is by no means a complete list, because users can define new defining definitions by the **CREATE**...**;CODE** and **CREATE**... **DOES>** structures (in 79-Standard dialect). When one creates a defining definition, he constructs both a compiler and an inner interpreter for a class of definitions to be defined. Let me try another way to express it, as in figure three.

The new compiler describes how each of the new definitions is to be constructed or compiled into the dictionary. The new inner interpreter, written in machine code, will execute or interpret the new definition when it is finally invoked and executed. The **CREATE**... **DOES>** structure allows the user to define the inner interpreter in high-level words similar to those used in a regular colon definition. How the high-level inner interpreter works depends upon the implementation, but its function is similar to an inner interpreter defined entirely in machine code.

A couple of examples probably will be helpful. Please refer to figure four. **MSG** and **ARRAY** are thus two new defining words in our Forth system. When these two words are executed in their appropriate contexts, they will compile new definitions into the dictionary:

MSG HELLO HOW ARE YOU?"
MSG ANSWER I AM FINE. AND YOU?"
10 ARRAY VECTOR

When **MSG** or **ARRAY** is executed, the compiler part of their definitions constructs new definitions on top of the dictionary. When the new definitions created by **MSG** or **VECTOR** are executed, the

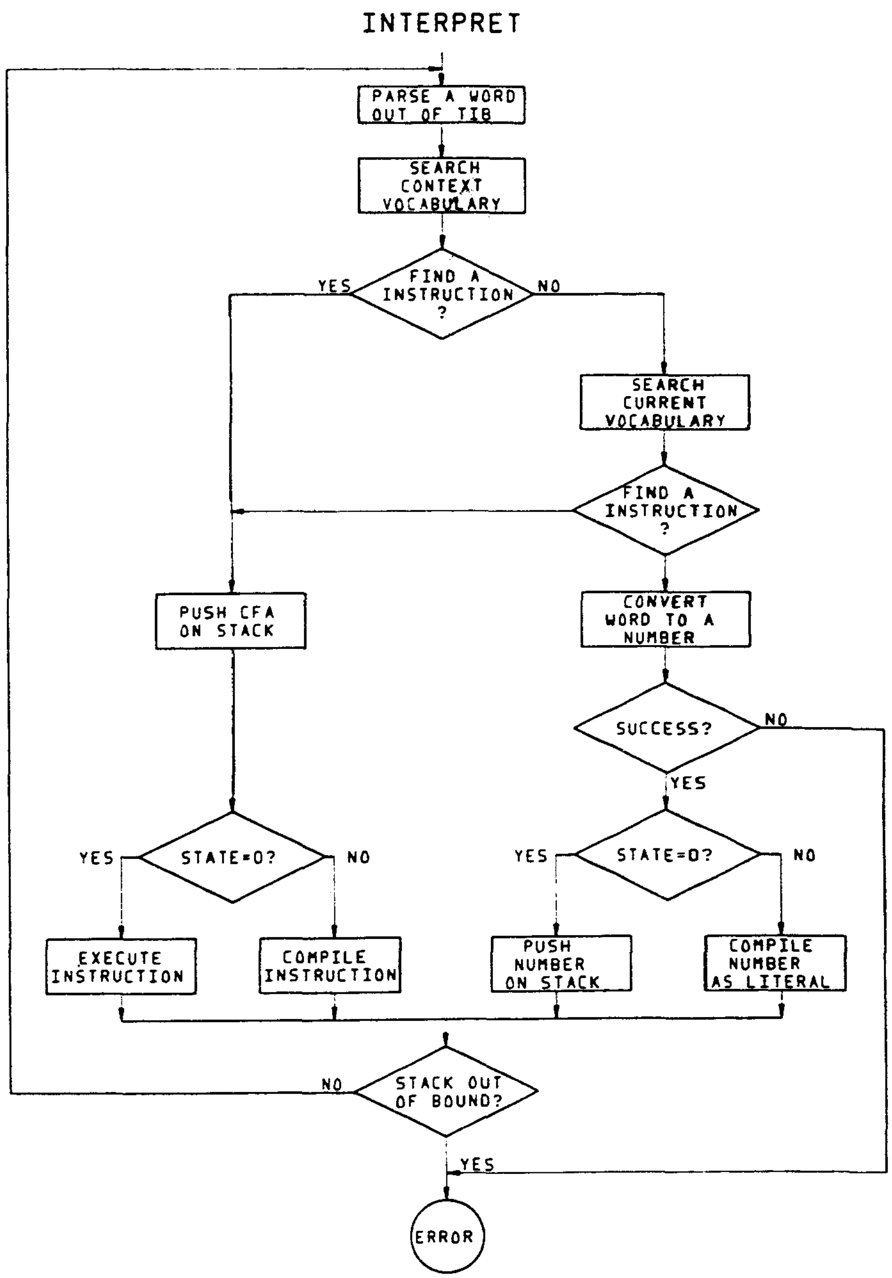


Figure Two
Actions taken by the test interpreter

: <name> CREATE <a new compiler>
;CODE <an inner interpreter in machine code>
: <name> CREATE <a new compiler>
DOES> <an inner interpreter in Forth words>

Figure Three
Creating defining definitions

```
: MSG CREATE                    ( build the header)
      34 WORD  C@ 1+ ALLOT      ( compile the message)
                                ( That's the compiler!)
      DOES>                     ( Here comes the interpreter)
      COUNT TYPE ;              ( print the message.)
: ARRAY CREATE                  ( build the header)
         HERE OVER ERASE        ( initialize the array to 0)
         ALLOT                  ( allocate array memory)
         DOES>                  ( end of compiler)
         SWAP 2* + ;            ( The inner interpreter returns)
                                ( the address of an element)
                                ( in the array.)
```

Figure Four
Defining inner interpreters in high-level Forth

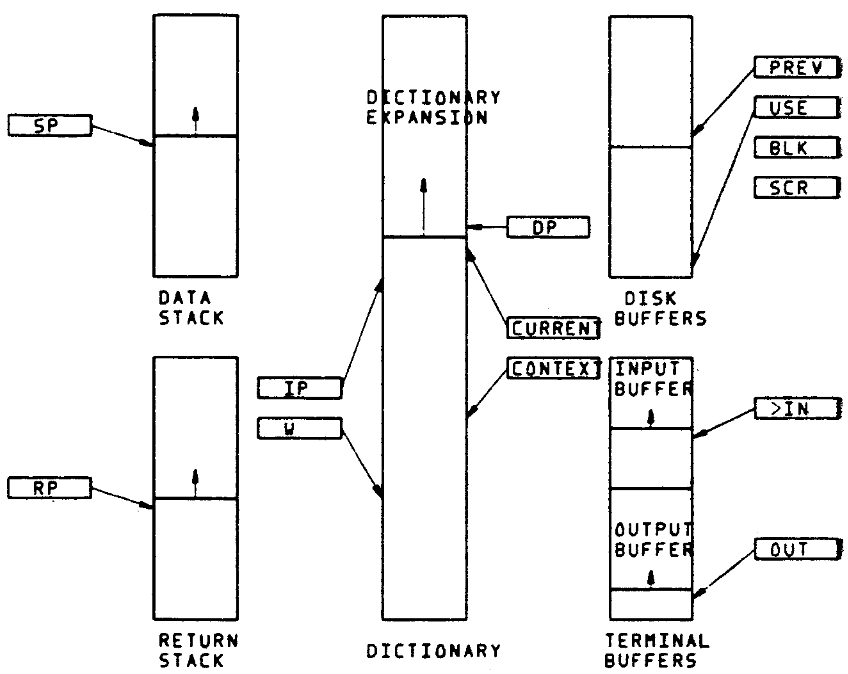


Figure Five
The Forth Virtual Computer

interpreter part of **MSG** or **VECTOR** is invoked to "interpret" the data stored in the parameter fields of the new definitions. In the cases of **HELLO** and **ANSWER**, strings compiled into the parameter fields are printed out on the terminal. In the case of **VECTOR**, the address of an element in the array is placed on the stack.

I hope these examples illustrate the intended functions of a defining definition: compiling new definitions and interpreting them when a new definition is invoked. When we program in Forth, normally we add new definitions to the Forth system using the pre-defined defining words like :, **CODE**, **CONSTANT**, **VARIABLE**, and **VOCABULARY**. This capability, according to Kim Harris[2], is "Forth extensibility of the first kind." This capability, though extremely powerful, limits us to these pre-defined data structures. The ability to create new types of defining words which in turn generate new classes of commands and data structures is "Forth extensibility of the second kind." This ability sets Forth well above any other existing high-level programming language, because it provides us with a very simple tool to build customized compilers and interpreters for our specific applications. Many examples have appeared in Forth literature based upon this compiler/interpreter construction, like regular and cross assemblers, meta-compilers, data-base systems, file management, floating-point and extended-precision data structures.

Virtual Forth Computer

In the preceding paragraphs, we touched upon the function of **EXECUTE** and that it can invoke an inner interpreter to perform the actions desired of the command to be executed. How does **EXECUTE** do this kind of magic? How does the inner interpreter carry on from there?

It is not an easy job to explain the internal actions in a Forth computer. I have seen lots of people bogged down in Chapter 9 of *Starting Forth*[3] for months, not able to fully grasp the sequence of events in the exeuction of a high-level Forth command. Pointers are moved around and control is jumping from one level to another. Many times I myself got lost trying to trace the sequence for my students. It seemed to be a hopeless task.

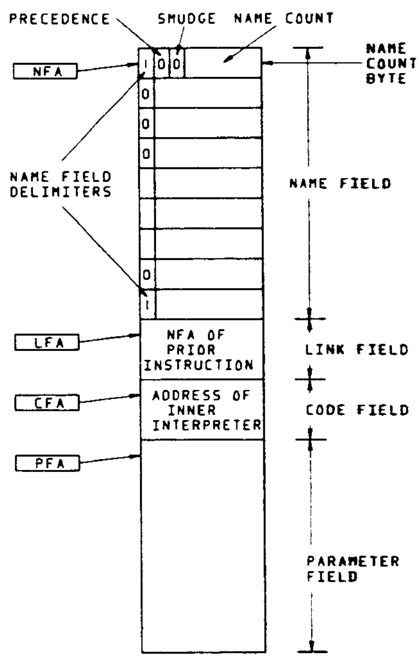


Figure Six
Instructions Format

EXECUTE: (cfa ---)		Code field address on data stack.
MOV	(SP),W	Copy top of data stack, cfa, to **W** register.
INC	SP	Pop data stack.
MOV	(W),PC	Copy the address of inner interpreter in the code field into **PC**. The inner interpreter will be executed next. **W** remains pointing to the cfa of the Forth word under processing.

Figure Seven
The execution of EXECUTE

NEXT:	**IP** points to the next word to be executed.
MOV (IP),W	Copy contents of **IP**, the cfa of the next word to be executed, to **W** register.
INC IP	Point **IP** to the word after the next word to continue the execution sequence.
MOV (W),PC	Execute the inner interpreter whose address is now in **W**, taken from the code field of the current word under processing.

Figure Eight
How NEXT operates

The only way to convey some sense of rationality is to bite the bullet and present the fundamental processes in the Forth virtual computer. It may not communicate full understanding of the Forth system, but at least it will not add to the feeling of confusion, chaos and helplessness.

The virtual Forth computer is a program loaded into the memory of a real computer. It partitions the computer memory into areas of specific function and enables the real computer to process Forth command streams. Figure five is a schematical representation of the functional parts in a virtual Forth computer. It consists of a dictionary, two stacks, a terminal input buffer and a number of disk buffers.

The virtual Forth computer uses a set of registers to store the most vital information for control of the flow of execution sequences. They are:

SP	Data Stack Pointer
RP	Return Stack Pointer
IP	Interpretive Pointer
W	Current Word Pointer
PC	Program Counter

The program counter **PC** and the return stack pointer **R** are usually registers in the host CPU. The data stack pointer **SP**, the interpretive pointer **IP** and the current word pointer **W** can reside in memory if the host CPU does not have enough registers.

To express the functions precisely, I shall use a "universal assembler" with three instructions:

MOV	src,dest	Move data from src to dest.
INC	dest	Increment contents of dest.
DEC	dest	Decrement contents of dest.

Parentheses around the src or dest indicate "the contents of," a level of indirection.

The dictionary is a linked list of word definitions. Each word definition consists of four fields, as shown in figure six. The name field and the link field allow definitions to be linked into a linear list which can be searched by the text interpreter. The code field contains the address of the inner interpreter for this definition and the parameter field contains necessary information specific to the task defined for this definition. Our attention will be concentrated in the code field and the parameter field.

The Code Interpreter

Where shall we start to explain the inner interpreters? The best place to start is probably the **EXECUTE** in the text interpreter, which invokes an inner interpreter to execute a command. We should note that when **EXECUTE** is executed, the code field address of the command to be executed is placed on the data stack by the text interpreter. See figure seven. **EXECUTE** moves the cfa into **W** register and jumps to the inner interpreter indirectly through **W**. Because the **W** register is still pointing to the code field of the current word, the inner interpreter can use the **W** register to access information contained in the parameter field of the current word for whatever purposes meant for the inner interpreter. Many Forth implementations increment the **W** register before jumping to the inner interpreter. They are called post-incrementing inner interpreters. This is convenient because the **W** register will then point squarely at the parameter field where information is to be retrieved. My "universal assembler" would be very messy if I had to increment **W** before the jump; therefore, I will let the inner interpreters do the incrementing if they need to do it.

All inner interpreters must end their execution process with a code sequence named **NEXT**, which returns control to the text interpreter. If an inner interpreter is called by another high-level word, control will be returned to the calling high-level word by **NEXT**. The presumption of **NEXT** is that the address of the next word in the execution sequence is contained in the interpretive register **IP**, which is used by the address interpreter (allow me to get ahead of myself for a short moment), to scan a list of addresses as compiled in the parameter field of a colon definition. **NEXT** operates as shown in figure eight.

All words in the dictionary can be invoked by **EXECUTE** if the code field address is pushed on the data stack, or by **NEXT** if the code field address, compiled in the dictionary, is pointed to by the **IP**.

I must emphasize this point: when a Forth word is executed, it is its inner interpreter which gets executed by the host computer. **NEXT** and **EXECUTE** get the address of the inner interpreter from the code field of the word definition.

Let us take a closer look at the code definitions, which are defined in the machine code of the host computer. In a code definition, the host machine code is contained in the parameter field of the definition. What is the inner interpreter for a code definition? Look at the contents of its code field. Guess what? The code field is pointing right at the parameter field, one cell after itself! Therefore, when **EXECUTE** or **NEXT** invoke this code definition, the machine code in the parameter field is executed by the host. Each code definition thus contains its own inner interpreter. From this point of view, it should be very obvious why every code definition must end with the **NEXT** code sequence. Though it is advantageous to think of each code definition, with its own inner interpreter, as a class by itself, our general practice is to group all code definitions together as one class of Forth words. After all, they still share the same compiler, which should be called an assembler. We can assign to them a fictitious common inner interpreter, **(.+2)**, a pointer to the parameter field, or the "code interpreter."

The Address Interpreter

In a colon definition, the parameter field contains a list of code field addresses. The inner interpreter for this class of Forth words must be able to scan this list of addresses and execute them in the appropriate sequence. This inner interpreter should be named properly the "address interpreter," in order not to be confused with other inner interpreters. In the fig-FORTH Model, this address interpreter is named **DOCOL** (figure nine). **DOCOL** uses the interpretive pointer **IP** in a way very similar to that in which the CPU uses the program counter **PC** to keep track of the execution sequence. **IP** scans through a list of code field addresses as the **PC** scans through a list of host machine instructions. If the code field address points to another colon definition, the **IP** must be used to scan a new list

```
DOCOL:            W register points to the code field of the current word
                  being executed.
  DEC  RP         Make room on the return stack.
  MOV  IP,(RP)    Push the address of the next word to be executed on the
                  return stack, because IP will be used to scan the address
                  list of the current word.
  INC  W          Point W to the parameter field of the current word, at the
                  head of the new address list.
  MOV  W,IP       IP is pointing to the head of the new address list, ready
                  for NEXT.
  MOV  (IP),W     Get the first code field address into W register.
  INC  IP         Move IP to the next address.
  MOV  (W),PC     Execute the first word in the address list of the colon
                  definition.
```

Figure Nine
The fig-FORTH address interpreter

```
EXIT:             The return address is saved on the return stack.
  MOV  (RP),IP    Restore IP from the return stack.
  INC  RP         Pop the return stack. Unnest by one level.
  MOV  (IP),W     This is the NEXT again.
  INC  IP
  MOV  (W),PC     Return to the caller.
```

Figure Ten
EXIT returns control to a calling definition

```
DOCON:            W register points to the code field of the constant.
  INC  W          Point W to the parameter field.
  DEC  SP         Make room on data stack.
  MOV  (W),(SP)   Push the contents of the parameter field onto the data stack.
  MOV  (IP),W     Constant function completed,
  INC  IP         get NEXT to move on.
  MOV  (W),PC

DOVAR:            W register points to code field.
  INC  W          W points to the parameter field.
  DEC  SP         Prepare a push.
  MOV  W,(SP)     Push the parameter field address onto the data stack, not
                  its contents as in DOCON.
  MOV  (IP),W     Call NEXT.
  INC  IP
  MOV  (W),PC
```

Figure Eleven
Interpreting constants and variables

of addresses. The old address in **IP** is preserved on the return stack so that **IP** can be freed to scan the new list. The return stack is thus an extension of the **IP** register, allowing a colon definition to call other colon definitions, which can then call other colon definitions. The nesting of colon definition calls is limited only by the depth of the return stack allocated in the virtual Forth computer.

At the end of a colon definition, control must be returned to the calling definition. The return address was saved on the return stack by **DOCOL**. The Forth word which returns control to the calling definition is **EXIT** (figure ten). **EXECUTE** and **NEXT** are analogous to the machine-code level **CALL** and **RTN** instructions, and **DOCOL** and **EXIT** are analogous to the **SUBROUTINE** and **RETURN** commands in Fortran or other high-level languages. They are the most important tools by which the virtual Forth computer finds its way through most of the Forth commands.

Constant Interpreter and Variable Interpreter

Code definitions and colon definitions are only two classes of Forth words among many other classes. In a regular Forth system, at least two more classes are provided: constants and variables. Their respective inner interpreters are **DOCON** (the constant interpreter) and **DOVAR** (the variable interpreter). Their functions as described by my universal assembler are shown in figure eleven.

Constants and variables are very similar in their structures. There is only one cell reserved in their parameter fields, and the numeric value of the constant or variable is stored in this cell. The only difference in their respective inner interpreters is that the constant interpreter pushes the contents of this cell onto the data stack while the variable interpreter pushes its address onto the stack.

Because constants and variables are used very often in Forth, their interpreters are usually defined in host machine code. The defining words **CONSTANT** and **VARIABLE** are themselves defined with the **CREATE . . . ;CODE** construct as in figure twelve.

Continued on page 35

Automatic Capitalization in Forth

Jeffrey B. Lotspiech
Thomas M. Ruehle
Boulder, Colorado

Until recently computer programmers entered their programs in all capital letters as a matter of necessity; keypunches and teletypes simply did not provide lower-case letters. Technology has progressed, but many programming languages (and some programmers) have retained an *upper-case mentality*. This is unfortunate; because readers are accustomed to reading text that is predominantly lower case, they read it more easily. Many programmers recognize this and will tolerate some inconveniences in order to enter their comments in both upper case and lower case, even if their programming language requires all other statements to be in upper case.

But it would be wrong to think that lower case belongs to comments only; variable names and action words can also benefit from both upper case and lower case, and especially from the visual contrast between lower case and upper case within a program. For example, our highest-level application programs in Forth typically consist of mnemonic high-level words nestled in **IF ELSE THEN** or looping control structures. Unfortunately, some stack management words usually creep in, even at the highest level; so the reader must suffer through an occasional **DUP**, **SWAP**, **DROP** or **OVER**. If the high-level, applications-oriented words are in lower case while the branching and stack management words are in upper case, the reader is provided with one more visual clue to help him decipher what is going on—certainly not a replacement for the other visual clues of indenting and spacing, but instead a nice complement to them. In fact, case should be considered another tool in the professional programmer's toolbox to help him with one of his major goals: making his programs clear and understandable.

Forth, of course, allows any characters except the blank and the null to be used in names; so there are no inherent restrictions on lower-case names. Since all the nucleus words are capitalized, however, you can quickly wear out your shift key and your patience if you try to write a program that effectively uses lower case. Even if your computer supports a CAPS LOCK key, remembering to turn that key on and off can be an annoyance and can even discourage proper commenting of code.

A possible solution is to ignore the case of alphabetic characters by changing **WORD** to fully capitalize every word it parses. Then you would enter your program entirely in lower case, and you would not worry about capital letters at all. This method is unsatisfactory because you lose the visual impact of having some words in upper case and some words in lower case.

Our solution is to type all of the program in lower case, as before, but to have the compiler capitalize a word permanently if it is "appropriate" for that word. And it is "appropriate" to permanently change a word to capitals if the compiler *cannot* find it as it was typed, but *can* find it in its capitalized version.

The screens in figures one through four show our implementation of this idea. The word **re-FIND** (figure two) is our new version of the standard nucleus word **-FIND** that, if necessary, will capitalize a word and try to find it again. Specifically, if the first **-FIND in re-FIND** fails to find a word, then **re-FIND**:

1. Backs up the input stream by one word using **unWORD**.
2. Finds the new memory address of the input stream using **INmemory**.
3. Capitalizes the input word in memory using **CAPITALIZE**.
4. Sends **-FIND** to search for the newly capitalized word.
5. Keeps the change permanently on disk using **keep**, if the new word is found.

Perhaps the only thing about this process that is not completely straightforward occurs in **unWORD**, whose job it is to *undo* the action of the nucleus word **WORD**. **WORD** normally advances the input pointer **IN** past the trailing delimiter of the word it parses (a blank, when **WORD** is called by **-FIND**). Thus, **unWORD** adds one to the length of the word (found at **HERE**) to skip that delimiter. However, **WORD** (actually **ENCLOSE** called by **WORD**) treats an ASCII null as a special case delimiter. It will *never* skip an ASCII null under any circumstances. Therefore, if the word parsed was delimited by a null, **unWORD** would back up one character too many. The precise solution to this problem is to have **unWORD** examine the character at **IN** and the character immediately before it, and if the character at **IN** is a null and the character before is not a blank, back up one character less. Our simpler solution is merely to make sure that **IN** does not go below zero, which is the only serious problem that could occur. (This forces the programmer to leave a blank after a closing double quotation mark or after a closing parenthesis if the next word is delimited by a null and he wants to have it automatically capitalized. Because that blank should be left there for readability anyway, and because words on disk screens rarely end up delimited by a null in any case, this restriction is inconsequential.)

The word **reFIND** can be used anyplace that **-FIND** is normally used, except in **CREATE**. The **-FIND** in **CREATE** expects *not* to find its target word (the word being created); if you capitalized and tried again, every newly created word in the dictionary would end up capitalized. Actually, the only place we have used **re-FIND** is in **INTERPRET**. We call this new version of the interpreter **interpret**, and it is shown in figure three. **[COMPILE] FORGET**, and ' (a tick) could also use **re-FIND**, but we are satisfied to require programmers using these words to type the word which follows in the correct case.

The new interpreter runs more slowly, because some words it encounters will require two searches: one for the original version of the word and one for the capitalized version. The first compilation of a screen permanently capitalizes those words that need to be capitalized. The only words that are not found in one

```
: CAPITALIZE ( addr count -- )  ( capitalizes a string )
   OVER  +  SWAP  DO
        I  C@  DUP  [ HEX ] 60  >  OVER  7B  <  AND
        IF  [ FF 20 - ] LITERAL  AND  ENDIF     I  C!
   LOOP ;                              DECIMAL

: INmemory ( -- addr )  ( address where input will come from )
   BLK @ IF       BLK @ BLOCK
         ELSE     TIB @            ENDIF     IN @  +  ;

: unWORD ( -- count )        ( backs up input over last word )
   IN @  DUP  HERE C@  1+  -    ( back up word length plus 1 )
   0  MAX  DUP  IN !  -   ;     ( but never past start of buf )
                  -->
```

Figure One
Automatic capitalization using CAPITALIZE
INmemory and unWORD

```
: keep  (  --  )  ( makes change permanent)
   BLK @    IF  UPDATE  ENDIF    ;

: re-FIND ( flag -- [ pfa len ] flag )  ( capitalizing -FIND )
  -FIND  -DUP  0=
  IF  ( not found as is )
      unWORD  INmemory  SWAP  CAPITALIZE  ( make uppercase  )
      -FIND  DUP  IF    keep    ENDIF     ( keep if OK now  )
  ENDIF  ;
            -->
```

Figure Two
Automatic capitalization using
keep and re-FIND

```
: interpret ( -- )  ( the lowercase equivalent of INTERPRET )
  BEGIN  re-FIND ( re-FIND only difference from INTERPRET)
     IF ( FOUND, perhaps after CAPITALIZEing it )
        STATE @ <
        IF    CFA ,    ELSE    CFA EXECUTE     ENDIF
     ELSE  HERE  NUMBER  DPL  @   1+
        IF     [COMPILE] DLITERAL
        ELSE  DROP [COMPILE] LITERAL ENDIF
     ENDIF  ?STACK    AGAIN   ;            -->
```

Figure Three
Automatic capitalization using interpret

```
: ZAP ( oldpfa newpfa -- )  ( replaces word in nucleus )
   CFA  OVER  !                  ( "compile" newpfa into old    )
   [  ' ;S  CFA  ]      LITERAL  OVER  2+   !      ( force ;S    )
   [  ' ;   CFA  @  ]  LITERAL  SWAP  CFA  !  ;  ( force DOCOL)

  ' INTERPRET    ' interpret    ZAP
```

Figure Four
ZAP—nucleus patching word

search when the screen is re-compiled are literal numbers (which are not found in any search). Therefore, a literal number takes twice as long to compile with our new interpreter as it takes with the standard interpreter. This could be fixed (for example, **CAPITALIZE** could check to see if it really capitalized anything), but we find it does not create enough of a problem in compilation time to concern us.

In theory, we would re-compile our new version of the interpreter into the nucleus. However, like many Forth users, we do not have a Forth nucleus that we can directly boot up (we must cross-compile it from an IBM System/370 host processor). So, we have invented a second word, **ZAP**, used for testing. It can quickly replace a selected word in the nucleus with a new word defined outside the nucleus. **ZAP** is shown in figure four. It takes the parameter field of the old word and compiles a two-word sequence into it: (new CFA, **;S**). This has the effect of branching references to the old word in the nucleus with a new word defined outside the nucleus. **ZAP** stores the address of **DOCOL** in the code field of the old word, so that if the old word was not originally a colon definition, it becomes one after being zapped. Since **ZAP** re-writes the first four bytes of the parameter field of the old word, this field must be at least four bytes long. This criterion usually eliminates words defined with **USER**, **VARIABLE** or **CONSTANT** as candidates to be zapped.

Notice that when the screen shown in figure four is loaded, **INTERPRET** will interpret a **ZAP** to change itself to its new version in the middle of its execution, and will resume executing with the new logic as if nothing had happened. To be honest, it is more a matter of luck than of design that this works (in fig-FORTH systems); however, it does indicate some of the power and usefulness of **ZAP**. We are also confident that any programmer who is serious about writing readable code (and has been using lower case toward this end) will find automatic capitalization a welcome addition to the professional programmer's toolbox.

More Screens for the Apple

Allen Anway
Superior, Wisconsin

After using an Apple fig-Forth public-domain disk by George B. Lyons and updated by Mark R. Abbott, I got an Apple IIe. Wouldn't it be nice to use the extra 16K memory (total 64K) for screen storage, thus freeing regular RAM? Ah, but it's bank switched over ROM and some routines use ROM. The solution is to download 1K blocks of the upper 16K into regular RAM (below $C000; see memory map in figure one) at a fixed 1K block. The following screens perform this automatically, even permitting over-writing and up-dating.

I use the notation shown in figure two for commenting the block programs. Think of J as the square root of -1 for "imaginary" bank-switched RAM. The common Forth words **USE**, **PREV**, **+BUF**, **EMPTY-BUFFERS**, **DR0**, **DR1**, **WBUF**, **BUFFER**, **BLOCK** and **FLUSH** retain their original meanings, but with a little more apparatus to download and upload. However, watch out for **BANK**. It is very specific.

Allen Anway is the Director of Instructional Computing at the University of Wisconsin, Superior.

```
SCR # 40
( SCREEN # 040 ) ( 1-19-84 DISK 1   AA )
HEX            ( ALLEN ANWAY UW-SUPERIOR )
CODE BON1 C08A LDA, C083 LDA, C083 LDA,
 NEXT JMP,                       ( --- )
CODE BON2 C082 LDA, C08B LDA, C08B LDA,
 NEXT JMP,                       ( --- )
CODE BOFF C082 LDA, C08A LDA, NEXT JMP,
                                 ( --- )
: BANK ( NA --- BAJ ) 1E AND DUP 8 < IF
   BON1 8 OR ELSE BON2 ENDIF 200 * C000
   OR ;  ( CONV NA & TURN-ON J RAM )
   0 VARIABLE USE      0 VARIABLE PREV
B380 CONSTANT BADD
B782 CONSTANT FIRST B7A2 CONSTANT LIMIT
: +BUF ( NA --- NA'\FLAG ) 2+ DUP LIMIT
   = IF DROP FIRST ENDIF DUP PREV @ - ;
: UPDATE  ( --- ) PREV @ @ 8000 OR PREV
   @ ! ;
-->

SCR # 41
( SCREEN # 041 ) ( 1-19-84 DISK 2   AA )
: EMPTY-BUFFERS ( --- ) FIRST DUP PREV !
   DUP USE ! 2- LIMIT OVER - ERASE ;
   ( ALSO SETS BLOCK TERM TO $0000 )
: EBUFS EMPTY-BUFFERS ;          ( --- )
: DR0      0 OFFSET ! ;          ( --- )
: DR1 BLK/DR OFFSET ! ;          ( --- )
: PR>UP ( --- ) ( REG RAM TO J   RAM )
   BADD PREV @ BANK B/BUF CMOVE BOFF ;
: PR<DN ( --- ) (   J RAM TO REG RAM )
   PREV @ BANK BADD B/BUF CMOVE BOFF ;
: WBUF ( NA --- ) ( >DISK, DE-UPDATE )
   >R R BANK R @ 7FFF AND DUP R> !
    0   R/W  BOFF  ;
-->
```

```
SCR # 42
( SCREEN # 042 ) ( 1-19-84 DISK 3   AA )

: BUFFER ( B# --- BAJ )
  USE @ DUP >R BEGIN +BUF UNTIL USE !
  R    @ 0< IF R      WBUF ENDIF
  R    ! ( B# INTO OLD USE )
  R    PREV !    R> BANK ;

: BLOCK   ( B# --- BA   )
  OFFSET @ + >R PREV @ DUP @ R - DUP +
  IF PR>UP
      BEGIN +BUF 0=
         IF DROP R BUFFER R 1 R/W
                           PREV @ ENDIF
         DUP @ R - DUP + 0= UNTIL
      DUP PREV ! PR<DN ENDIF
  RDROP DROP BADD ;

: FLUSH   (      ---       )
  PR>UP PREV @ BEGIN DUP @ 0< IF
  DUP WBUF ENDIF +BUF 0= UNTIL DROP ;
```

```
Memory map:

$B380 - B77F  BADD   "permanent" single block area that is down-loaded
$B780 - B781         $0000 block terminator
$B782 - B7A1  FIRST  name address  area of $20 bytes
$B7A2 - B7FF  LIMIT  user variable area of $5E bytes, this is also UP @
$B800 - BFFF         DOS          area

bank2  bank1         switches for bank selects
$C082  $C08A         enable ROM, write-protect RAM
$C083  $C08B         enable RAM, two accesses enable read/write

$D000 - DFFF         ROM, bank-switched RAM, RAM
$E000 - FFFF         ROM, bank-switched RAM
```

Figure One

```
NA   name  address ... address of name of block, see USE and PREV
N    name          ... NA @      name of block = block#.OR.8000 if updated
BA   block address ... address of block contents, regular RAM, see BADD
BAJ  block address ... address of block at bank-switched RAM
B#   block number  ... 1 - 140 for DR0, 141 - 280 for DR1
```

Figure Two

Interactive Editing

Wendall C. Gates
Santa Cruz, California

Tom Blakeslee's article "Debugging From a Full-Screen Editor" in *Forth Dimensions* (V/2) described a novel and effective way to use a screen editor. His word **STEP** allows the execution of one Forth word at a time from within the full-screen editor, with a display of the stack at each step; putting the required values on the stack is done before entering the editor.

This concept can be further expanded. First, the ability to manipulate stacks (and other operations) can be brought conveniently inside the editor. The nested **INTERPRET** technique used in **STEP** is expanded to input and execute one line of Forth, from a convenient location on the screen.

A second desirable feature is for an aborted execution to return to the editor. With a simple nested **INTERPRET** as used in **STEP**, aborted execution results in writing the error message across the editing screen and aborting out of the editor as well. One method of automatically returning into the editor is shown in the accompanying screens (written in fig-Forth).

In screen #176, **DO.FORTH** moves the cursor to the bottom line of the monitor screen and clears the line; it then saves the current position in the editor and proceeds to accept one line of Forth, using **QUERY**. Execution is by **ED.INTERPRET** except it also saves the return stack pointer into a variable **ED.RPO** and it uses **ED.NUMBER** instead of **NUMBER**. **ED.NUMBER** (not shown) is **NUMBER** but with **?ERROR** replaced with **ED.?ERROR**. In turn, **ED.?ERROR** mirrors **?ERROR** but with **ERROR** replaced by **ED.ERROR** which is shown in screen #173. **ED.ERROR** prints the name of the token which caused the problem followed by a "?" and then reloads the return stack pointer with the address stored in **ED.RPO** instead of the original return stack origin. It then unnests back into the editor. This method won't recover from a system crash but it does minimize the inconvenience of typos, of entering hex numbers in decimal, and such. **DO.FORTH** is patched into the editor as a keyboard-executable command.

Substituting **ED.INTERPRET** for **INTERPRET** in Blakeslee's word **STEP** produces a similar result. However, **ED.ERROR** must test for current cursor condition and move to the last line on the display screen if not there already -- otherwise the error message will be written across the editing screen.

Legibly packing a listed screen with ID, editor instructions, executable Forth line and a stack display (in both hex and decimal) all on one 24 x 80 screen is a bit of a challenge. My arrangement is shown around screen #176 in figure one. The top five values (word values) on the computational stack are displayed and updated with each execution. The bottom line is the space for the executable Forth line and its response. This display arrangement requires replacing **LIST** with (what else?) **ED.LIST** which lacks the first **CR** of the usual version.

I call this technique "interactive editing." Using it produces that same euphoric feeling of power and control that comes the first time one uses a full-screen editor instead of a line editor. Stepping through code, one token at a time, while using the one-line execution to lift loop indices, do return stack operations, check variables, etc., with the stack display keeping itself current, enormously simplifies both hardware and software debugging. The technique is also a marvelous teaching tool. Try interactive editing — you'll like it!

Wendall C. Gates is President of Advanced Instrumentation Inc., where he supervises and participates in the hardware design and applications programming of instruments for pollution control and environmental monitoring.

```
Editing Screen # 176
  0 \ AI Screen Editor -- execute 1 line of FORTH        WCG 2.15.84
  1 : DO.FORTH
  2   0 23 GOTOXY  CLREOL       \ cursor to bottom line & clear it
  3   IN @ >R  BLK @ >R         \ save interpreting location in editor
  4   0 IN !    0 BLK !         \ initialize new interpret mode
  5   QUERY  ED.INTERPRET       \ input and execute 1 line of FORTH
  6   R> BLK !  R>  IN !        \ go back to editor
  7   .CUR ;                    \ restore cursor position on display
  8
  9   -->
 10
 11
 12
 13
 14
 15

^P = up     ^O = open line    ^Q = erase screen                 3 :    3h
^N = down   ^C = close line   ^X = execute word                 2 :    2h
^F = -->    ^R = insert spc   ^T = 1 line FORTH                 1 :    1h
^B = <--    ^\ = back tab    ESC = exit editor                 10 :    Ah
^D = DEL   TAB = fwd  tab  ^K,^Y = CLREOL, yankback
HEX  A 1 2 3 4 5  . .  5 4
```

Figure One
The interactive editor at work

```
SCR # 173
\ AI Screen Editor -- RP.! ED.RP0 ED.ERROR         WCG 3.01.84
HEX
CODE RP.!
  1909 , A229 , 09B2 , 2929 , DC  C, \ TOS to retn stk pointer
                                      \ coded for 1802
0 VARIABLE ED.RP0               \ storage for retn stack pointer
DECIMAL

: ED.ERROR
  CURRENT.LINE 15 =             \ cursor at bottom line?
    IF 0 23 GOTOXY THEN         \ if not, put it there
  HERE COUNT TYPE ."  ?"        \ print offending name with ?
  ED.RP0 @ RP.! ;               \ load retn stk pointer back to
                                \ previous position in editor
  -->


SCR # 175
\ AI Screen Editor -- ED.INTERPRET                WCG  2.20.84
: ED.INTERPRET
  RP@  2 -  ED.RP0 !                 \ save return stack pointer
  BEGIN
    -FIND IF STATE @ < IF CFA , ELSE CFA EXECUTE THEN
            ?STACK
            ELSE HERE ED.NUMBER DPL @ 1+
              IF [COMPILE] DLITERAL
                ELSE DROP [COMPILE]  LITERAL
                THEN
              ?STACK
            THEN
   AGAIN ;
  -->
 ED.INTERPRET is identical to INTERPRET, except saves index of
 return stack, and uses ED.NUMBER instead of NUMBER.
```

Parnas' it . . . ti Structure

Kurt W. Luoto
Redwood City, California

David Parnas proposes[1] a new control structure for the specification and implementation of algorithms. It is general enough that it includes **IF THEN**, **BEGIN UNTIL**, **BEGIN WHILE REPEAT** and **CASE**-like structures as special cases. In many cases it can simplify the expression of an algorithm, often resulting in more efficient programs. Often a single instance of the structure can replace several of the more restricted structures mentioned above, or eliminate having to define extra variables to carry around termination status in loops. The structure is useful for describing both deterministic and non-deterministic algorithms. Readers interested in the more formal aspects of this are encouraged to read this article.

This article gives a practical adaptation of this structure to Forth and gives an example of its use. Parnas uses the notation **it ti** (short for **it**eration) for this structure, along with ALGOL-like notation in his article. I have tried to use names more in the style of current Forth usage. One of the problems in thinking up names is that most of the good ones have already been taken, such as **IF** and **REPEAT**. I would be glad to see suggestions for improvements here.

The Structure

The structure is best introduced in graduated steps. The structure is **CASE**-like in appearance, the form being

IT <case> <case> . . .<case> **ENDIT**

(Although I refer to the individual clauses inside the structure as cases, this is not to be confused with the **CASE** structure.) Each case within the structure is either of the form

<condition> **IFF** <body> **BREAK**

or

<condition> **IFF** <body> **CONTINUE**

Figure one shows how a typical instance of the **IT ENDIT** structure might look. Here <condition> denotes any body of code.

When the **IT ENDIT** structure is encountered, the <condition> code of the first case is executed. If a true value is on the top of the stack when the **IFF** is encountered, then the <body> of that case is executed. (**IFF** always removes the flag from the top of the stack, just like the **IF** of an **IF THEN** structure.) If a false value is on the top of the stack when the **IFF** is encountered, then execution passes to the next case, where the process is repeated until the <condition> code for some case leaves a true value on the stack, whereupon the <body> of that case is executed. If no <condition> leaves true on the stack for any case, then execution passes to the code following **ENDIT**, i.e., the **IT** is terminated.

When the body of a case is executed, where it goes next is determined by the word that terminates the case. For a case ending in **BREAK**, execution passes to the code following **ENDIT**, i.e. the **IT** is terminated. For a case ending in **CONTINUE**, execution passes to the code following **IT**, i.e. the **IT** is repeated.

There are a couple of convenient conventions that I will use. The first is that the <condition> **IFF** portion of a case may be omitted, whereupon the body of the case will be executed unconditionally. Note that this only makes sense for the last case in an **IT ENDIT** structure. The other convention is that if the terminator for the last case is **BREAK**, it may be omitted, i.e. **BREAK ENDIT** is logically the same as **ENDIT**. In this implementation, this actually produces more efficient code.

Given these words, you could define logical equivalents of the other structures in terms of them as shown in screen 175. In this sense, the other structures are special cases of **IT ENDIT**. However, you would not actually implement the others this way in this particular implementation because the compile time would be longer and, for some of them, the code produced would be less efficient.

COR and CAND

While not part of the **IT ENDIT** structure *per se*, there are two very useful words that fit nicely into this implementation, **COR** and **CAND** (Conditional *OR* and Conditional *AND*). Normally, if two conditions must be tested for truth for a particular case in an **IT ENDIT** structure, the form would be

<cond 1> <cond 2> **AND IFF** . . .

But a problem arises if the second condition is dependent on the first. For example, if the first condition tests that a particular address is valid (e.g. non-zero) and the second tests the value stored at that address, it does not make sense to test the second condition if the first is not true. You could get around this by using.

<cond 1> **DUP IF** <cond 2>
AND THEN IFF . . .

```
IT
    <condition> IFF <body> BREAK
    <condition> IFF <body> CONTINUE
    <condition> IFF <body> CONTINUE
                          .
                          .
                          .
    <condition> IFF <body> BREAK
    <condition> IFF <body> CONTINUE
ENDIT
```

Figure One
A typical IT ENDIT Structure

instead. But the case could be more efficiently rewritten using **CAND** as

<cond 1> **CAND** <cond 2> **IFF** . . .

If, after <cond 1> is executed, a true value is left on the stack, execution proceeds with <cond 2>, otherwise a branch is made to the next case, i.e. <cond 2> is not executed.

The word **COR** behaves in complementary fashion. In a case of the form

<cond 1> **COR** <cond 2> **IFF** . . .

if, after <cond 1> is executed, a *false* value is left on the stack, execution proceeds with <cond 2>, otherwise a branch is made to the body of the case immediately following **IFF**, i.e. <cond 2> is not executed.

Both **CAND** and **COR** remove the flag from the top of the stack, just like the **IF** of an **IF THEN** structure. Any number of appearances of **CAND** and **COR**, in any order, may appear before an **IFF** within a case. **CAND** and **COR** provide a convenient way to deal with dependent conditions. They can also provide improved efficiency for independent conditions if the first condition is true (for **COR**) or false for (**CAND**) much more frequently than the second.

```
IT
                    .
                    .
                    .
    <cond 1> CAND <cond 2> IFF <body> BREAK
    <cond 1> CAND <cond 3> IFF <body> CONTINUE
    <cond 1> CAND <cond 4> IFF <body> CONTINUE
    <cond 1> CAND <cond 5> IFF <body> BREAK
    <cond 1> CAND <cond 6> IFF <body> CONTINUE
                    .
                    .
                    .
ENDIT
```

Figure Two
Testing repeatedly for a common condition

```
IT
                    .
                    .
                    .
  <cond 1> IFF  SUBCASES
                    <cond 2> IFF <body> BREAK
                    <cond 3> IFF <body> CONTINUE
                    <cond 4> IFF <body> CONTINUE
                    <cond 5> IFF <body> BREAK
                    <cond 6> IFF <body> CONTINUE
               ENDSUB
                    .
                    .
                    .
ENDIT
```

Figure Three
The most efficient SUBCASE and ENDSUB

Subcases

Oftentimes it occurs that several cases have compound conditions with a common condition between them, as shown in figure two. Here, it would be nice to be able to test the common condition only once. Since the Forth compiler typically cannot detect such repetitions, I have introduced the word-pair **SUBCASES ENDSUB** to provide a means of splitting up the body of a case into several subcases. Figure three shows how the example in figure two would be rewritten using these words.

Each subcase looks like a case in the **IT ENDIT**. As at the **IT ENDIT** level, each condition code of each subcase is executed until one leaves a true value on the stack, whereupon the body of the subcase is executed. A subcase terminating in **BREAK** branches to the code following **ENDIT** (not **ENDSUB**), just as a normal case does. A subcase terminating in **CONTINUE** branches to the code following **IT** (not **SUBCASES**), just as a normal case does. If none of the conditions are true for any of the subcases, then control passes to the next case, i.e. the code after **ENDSUB**. Notice that **ENDSUB** terminates the (super-)case. You should *not* put a **BREAK** or **CONTINUE** after the **ENDSUB** since it will be interpreted as the next case, consisting of an unconditional branch to the end or beginning, respectively, of the **IT ENDIT**.

I again use the convention that the <conditional> **IFF** in the last subcase in a list of subcases may be omitted. However, each subcase should end in either **BREAK** or **CONTINUE**.

Subcases may in turn have subcases.

Other Words

Screens 169 through 174 give an implementation of **IT ENDIT** in VLFORTH by Gerald Gutt of ROLM Corporation. In this particular implementation, the sequence **IFF BREAK** or **IFF CONTINUE** (a case with a null body) would generate two successive branch statements, the first being a conditional one around the second. This is somewhat inefficient, so I have included the words **IFF-BREAK** and **IFF-CONTINUE** to use in these situations. They generate more efficient code, but are otherwise optional. Not all implementations of **IT ENDIT** may need them.

An Example

Well-written Forth code tends to come in small, simple pieces so situations where this structure comes in handy are less frequent in Forth than in other languages. However, the same thing could be said about case structures. I believe this structure has its value even in Forth and deserves a permanent place in the Forth repertoire. It is not that you can't get by with only the other structures, but that **IT ENDIT** allows more natural expression of many algorithms. I think the following provides a good example.

Suppose that somewhere in your code, for some obscure reason, you want to test whether a string contains two commas separated by one or more characters. Or suppose you wanted to list all the words in the current vocabulary that have a particular number of characters and begin with three particular characters. One way to solve these and similar problems is with pattern-matching techniques. At the heart of the solution would be a word that would take two arguments, the first being a string that we wish to test and the second being a special string called a "pattern" that the first string is to be tested against. The word would return a "match" (true value) or "no match" (false value) according to the result of the test.

As a simple example, let our patterns be any string of ASCII characters. There are three special characters, — (dash), * (star) and ' (quote). All other characters are normal characters. Normal characters match themselves. The dash (—) matches any single character. The star (*) matches any string of characters (including the null string). The quote (') matches the character in the pattern following the quote (this allows us to specify dash, star and quote themselves as particular characters to be matched). A lone quote at the end of a pattern is ignored.

So, for example, the pattern ABC would match the string ABC. The pattern CON---- would match any string

Parnas' it . . . ti Structure

```
BLOCK: 169

( MISC. HELPING WORDS                       VLFORTH   KWL 20JAN84  )

: ROTB     ( N1 N2 N3 -- N3 N1 N2 )     ROT ROT    ;

: BACK!    ( TO-ADDRESS  BRANCH-ADDRESS  -- )       !    ;
( THIS WORD RESOLVES A BRANCH AT THE ADDRESS ON TOP OF THE    )
( STACK TO THE ADDRESS SECOND ON THE STACK.                   )
( THIS WORD IS SYSTEM DEPENDENT.  THIS SYSTEM USES ABSOLUTE   )
( ADDRESSES FOR BRANCHES.  SYSTEMS USING OFFSETS MIGHT HAVE   )
( A DEFINITION LIKE THE FOLLOWING:                            )
(  : BACK!   SWAP OVER -   SWAP !   ;                         )

: BACKFILL    ( TO-ADDRESS  LIST-ADDRESS  -- )
( THIS WORD RESOLVES A LINKED-LIST OF BRANCHES TO AN ADDR.    )
  BEGIN   DUP WHILE   DUP @ >R   OVER SWAP BACK!
  R>   REPEAT   DROP DROP   ;                                -->


BLOCK: 170

( IT..ENDIT WORDS                           VLFORTH   KWL 20JAN84  )

: IT    (  --   IT-LOCATION  ENDIT-LIST  IT-ID# )
  HERE   0   16   ;   IMMEDIATE
( YOU MAY USE ANY VALUE THAT DOES NOT CONFUSE THE COMPILER )
( IN PLACE OF 16, 17, 18 AND 19 IN THESE WORDS. THESE ARE  )
( SYSTEMS WITH "COMPILER SECURITY"                         )


: ENDIT  ( IT-LOCATION ENDIT-LIST IT-ID#                 OR )
         ( IT-LOC  ENDIT-LIST  CAND-LIST  IFF-ID# --        )
  ?COMP   DUP 18 = IF   DROP   HERE SWAP BACKFILL  16   THEN
  16 ?PAIRS   HERE SWAP BACKFILL   DROP   ;   IMMEDIATE

-->



BLOCK: 171

( IT..ENDIT WORDS                           VLFORTH   KWL 20JAN84  )

: CAND   ( IT-LOC  ENDIT-LIST IT-ID#                        OR )
         ( IT-LOC  ENDIT-LIST CAND-LIST COR-LIST CAND-ID#      )
         ( -- IT-LOC ENDIT-LIST CAND-LIST COR-LIST CAND-ID#    )
  ?COMP   COMPILE OBRANCH    DUP 17 = IF
     HERE   ROT ,   SWAP                 ELSE
     16 ?PAIRS   0 HERE   0 ,   17   THEN   ;   IMMEDIATE


: COR    ( IT-LOC  ENDIT-LIST IT-ID#                        OR )
         ( IT-LOC  ENDIT-LIST CAND-LIST COR-LIST CAND-ID#      )
         ( -- IT-LOC ENDIT-LIST CAND-LIST COR-LIST CAND-ID#    )
  ?COMP   COMPILE 0=   COMPILE OBRANCH   DUP 17 = IF
     ROT   HERE   SWAP ,   ROTB      ELSE
     16 ?PAIRS   HERE   0 ,   0 17   THEN   ;   IMMEDIATE  -->
```

```
BLOCK: 172

( IT..ENDIT WORDS                              VLFORTH   KWL 20JAN84  )

: IFF     ( IT-LOC  ENDIT-LIST IT-ID#                          OR     )
          ( IT-LOC  ENDIT-LIST CAND-LIST COR-LIST CAND-ID#            )
          (  --  IT-LOC  ENDIT-LIST  CAND-LIST  IFF-ID#               )
  [COMPILE] CAND   DROP   HERE ROT BACKFILL    18    ;   IMMEDIATE

: BREAK  ( IT-LOCATION ENDIT-LIST IT-ID#                       OR )
         ( IT-LOC  ENDIT-LIST  CAND-LIST   IFF-ID#                )
         (  --  IT-LOCATION  ENDIT-LIST   IT-ID#                  )
   ?COMP   DUP 16 = IF    2+    0 SWAP    THEN
   18 ?PAIRS    >R    COMPILE BRANCH    HERE SWAP ,
   HERE R> BACKFILL    16    ;    IMMEDIATE

-->

```

```
BLOCK: 173

( IT..ENDIT WORDS                              VLFORTH   KWL 20JAN84  )
: CONTINUE    ( IT-LOCATION ENDIT-LIST IT-ID#                    OR )
              ( IT-LOC  ENDIT-LIST  CAND-LIST  IFF-ID#              )
              (  --  IT-LOC  ENDIT-LIST  IT-ID#                     )
  ?COMP    DUP 16 = IF    2+    0 SWAP    THEN
  18 ?PAIRS    >R    COMPILE BRANCH    OVER HERE 0 , BACK!
  HERE R> BACKFILL    16    ;    IMMEDIATE

: SUBCASES    ( IT-LOC  ENDIT-LIST  CAND-LIST  IFF-ID#              )
          (  --  CAND-LIST SUB-ID#  IT-LOC  ENDIT-LIST  IT-ID#      )
  ?COMP    18 ?PAIRS    ROTB    19 ROTB    16    ;    IMMEDIATE

: ENDSUB      ( CAND-LIST SUB-ID#  IT-LOC  ENDIT-LIST  IT-ID#       )
              (  --  IT-LOC  ENDIT-LIST  IT-ID#                     )
  ?COMP    16 ?PAIRS    ROT 19 ?PAIRS
  ROT HERE SWAP BACKFILL    16    ;    IMMEDIATE               -->
```

```
BLOCK: 174

( IT..ENDIT WORDS                              VLFORTH   KWL 20JAN84  )

: IFF-CONTINUE
          ( IT-LOC  ENDIT-LIST IT-ID#                          OR     )
          ( IT-LOC  ENDIT-LIST CAND-LIST COR-LIST CAND-ID#            )
          (  --  IT-LOC  ENDIT-LIST  IT-ID#                           )
  [COMPILE] COR    DROP    HERE SWAP BACKFILL
  >R OVER R> BACKFILL    16    ;    IMMEDIATE

: IFF-BREAK
          ( IT-LOC  ENDIT-LIST IT-ID#                          OR     )
          ( IT-LOC  ENDIT-LIST CAND-LIST COR-LIST CAND-ID#            )
          (  --  IT-LOC  ENDIT-LIST  IT-ID#                           )
  [COMPILE] COR    DROP    HERE SWAP BACKFILL
  OVER    DUP IF    BEGIN    DUP @    WHILE    @    REPEAT    !
               ELSE    ROT    2DROP    THEN    16    ;    IMMEDIATE
```

beginning with CON and having exactly seven characters. The pattern Z* would match any string beginning with Z. The pattern *,—*,* would match any string containing two commas separated by one or more characters. The string '--- would match any three-character string beginning with a dash. You get the idea. More complicated systems of patterns can be made, but this example will suffice here. Think about how you would implement this using standard constructs.

The word **MATCH** defined in screens 176 through 178 takes two strings, each in the form of length-of-string and address-of-first-character, and returns a true (match) or false (no match) value. (The code can fit on one screen, but I have spread it out for better commenting and readability.) The algorithm I used keeps two pointers, initialized to the beginning of the pattern and the beginning of the string to be matched respectively. At each step or iteration in the algorithm, this series of conditions is tested:

1) If I have run out of both pattern and string (the pointers are at the end of both), then there is a match. Return true.

2) Otherwise, if I have run out of pattern but I have not run out of a string, then there is a mismatch. "Retry".

3) Otherwise, if the next character of the pattern is a star (*) and the star is also the last character in the pattern, then there is a match. Return true. (This check is really not necessary but makes the code more efficient for this common case.)

4) Otherwise, if the next character of the pattern is a star (*), then save the current string pointer and the pointer to the next character in the pattern. Go back to 1).

5) Otherwise, if the next character is a quote (') and it is the last character in the pattern, discard it by incrementing the pattern pointer. Go back to 1).

6) Otherwise, if there is no more string left (but there is more pattern), then there is a mismatch. "Retry".

7) Otherwise, if the next character in the pattern is a dash (—), *or* the next character of the pattern (or the one after the quote if the next pattern character is a quote) is the same as the next character of the string, then increment both pointers and go back to 1).

8) Otherwise, there is a mismatch. "Retry".

Actually there is another case at the beginning of the loop for handling part of the star's function. Those cases that end in "Retry" leave a true value on top of the stack for this case to test. The others leave a false value, as does the initialization. A true value indicates that a mismatch has occurred since the last, if any, occurrence of a star in the pattern. In this case I need to reset the string and pattern pointers to the values that were saved at the last star occurrence, increment the string pointer by one, save the new pointers and try matching again. If there was no star in the pattern previously, or if the saved string pointer is already at the end of the string, then the string definitely does not match the pattern and a false value is returned.

The word **MLIST**, defined in screen 179, takes a word from the input stream, and, treating it as a pattern, displays all the words in the current vocabulary whose names match that pattern. Its implementation may vary from system to system, depending on how you get from LFAs to NFAs, etc.

Summary

I have given a practical Forth implementation of Parnas' **it ti** structure and an example of its use. I hope this article has given some idea of the usefulness of Parnas' structure and will encourage further discussion. I have not covered all aspects of this structure here, such as implementations of the **init** predicate mentioned by Parnas in his article. This would be a word used within the body of an **IT ENDIT** structure that takes no arguments and returns a true value on the first iteration and a false value thereafter.

```
BLOCK: 175

( HYPOTHETICAL DEFINITIONS                VLFORTH   KWL 20JAN84  )
: IF        [COMPILE] IT    [COMPILE] IFF    ;    IMMEDIATE
: ELSE      [COMPILE] BREAK    ;    IMMEDIATE
: THEN      [COMPILE] ENDIT    ;    IMMEDIATE
: BEGIN     [COMPILE] IT    ;    IMMEDIATE
: WHILE     COMPILE 0=    [COMPILE] IFF    [COMPILE] BREAK    ;
  IMMEDIATE
: REPEAT    [COMPILE] CONTINUE    [COMPILE] ENDIT    ;    IMMEDIATE
: UNTIL     COMPILE 0=    [COMPILE] IFF    [COMPILE] CONTINUE
            [COMPILE] ENDIT    ;    IMMEDIATE
: AGAIN     [COMPILE] CONTINUE    [COMPILE] ENDIT    ;    IMMEDIATE
: INCASE    [COMPILE] IT    ; IMMEDIATE
: OF        COMPILE OVER    COMPILE =   [COMPILE] IFF
            COMPILE DROP    ;    IMMEDIATE
: ENDOF     [COMPILE] BREAK    ;    IMMEDIATE
: ENDCASE   [COMPILE] ENDIT    ;    IMMEDIATE
```

```
BLOCK: 176

( PATTERN MATCHING WORD                   VLFORTH   KWL 20JAN84  )
: MATCH    ( STR-ADDR STR-CNT PAT-ADDR PAT-CNT -- FLAG )

   ( CONVERT ADDRESSES AND COUNTS INTO FORM USABLE FOR SEARCH )
   OVER +    1 -    SWAP    >R >R    OVER +    1 -    SWAP
   R> SWAP >R    0 0    R> R>

   ( STACK:    STR-LIM   PAT-LIM   0   0   STR-ADDR   PAT-ADDR )

   ( LEAVE A ZERO IN ORDER TO SKIP "RETRY" CASE   )    0
   IT
      ( CHECK FOR "RETRY" CASE )
      IFF    SUBCASES
         3 PICK   0=    COR    6 PICK   3 PICK   U<    IFF    0 BREAK
         2DROP    SWAP 1+ SWAP    2DUP    0 CONTINUE    ENDSUB
-->
```

```
BLOCK: 177

( PATTERN MATCHING WORD                   VLFORTH   KWL 20JAN84  )

     ( CHECK FOR NO MORE PATTERN LEFT )
     5 PICK   OVER   U<    IFF    SUBCASES
        6 PICK    3 PICK   U<    IFF    1 BREAK
        1 CONTINUE    ENDSUB

      ( CHECK FOR STAR IN THE PATTERN )
      DUP C@   ASCII *   =    IFF    SUBCASES
         5 PICK   OVER   =    IFF    1 BREAK
         1+    ROT DROP ROT DROP    2DUP    0 CONTINUE    ENDSUB

     ( IGNORE A LONE SINGLE QUOTE AT THE END OF PATTERN )
     DUP C@    ASCII '   =    CAND
            5 PICK   OVER   =    IFF    1+    0 CONTINUE
-->
```

```
BLOCK: 178

( PATTERN MATCHING WORD                    VLFORTH   KWL 20JAN84  )
    ( CHECK FOR MORE STRING )
    6 PICK  3 PICK  U<   IFF   1 CONTINUE

    ( CHECK FOR DASH IN PATTERN )
    DUP C@   ASCII -  =   COR

       ( IF NO DASH, CHECK FOR SINGLE QUOTE IN PATTERN )
       DUP C@   ASCII '  =   IF   1+   THEN

       ( CHECK PATTERN CHAR. AGAINST STRING CHAR. )
       OVER C@  OVER C@  =   IFF   1+ SWAP  1+ SWAP  0 CONTINUE

    ( NONE OF THE ABOVE CASES. GO RETRY )
    1 CONTINUE
  ENDIT  ( LEAVE FLAG ONLY )   >R   2DROP 2DROP 2DROP   R>   ;
```

```
BLOCK: 179

( LIST VOCABULARY MATCHING PATTERN      VLFORTH   KWL 20JAN84  )

HEX

: MLIST   ( -- )
   ( TAKES A WORD FROM INPUT, TREATS IT AS A PATTERN, AND )
   ( SEARCHES FOR MATCHING VOCABULARY NAMES.               )
   CR   BL WORD DROP   CONTEXT @ @
   BEGIN   DUP NFA COUNT   1F AND   HERE COUNT   MATCH
      IF   DUP NFA .ID   SPACE   THEN
     @   DUP 0= UNTIL
   DROP   ;

DECIMAL
```

I have also not covered extensions of this to **DO LOOP** structures. Interested readers can easily implement these.

My special thanks to ROLM Corporation for use of their facilities in preparing this article.

References

1. Parnas, David L. "A Generalized Control Structure and Its Formal Definition." *Communications of the ACM* 26,8 (August 1983), 572-581.

Kurt W. Luoto is a programmer in Large Systems Engineering at ROLM Corporation. He heard of Forth's "audacious claims" in 1980, found them to be true and has maintained an interest ever since.

Letters *(Continued from page 7)*

A quick scan of this issue reveals several different type sizes for the presentation of screens. It is recognized that space in *Forth Dimensions* is at a premium and certainly smaller type, hopefully, means more articles. However, please try to standardize on a readable type size.

My thanks to the many contributors to *Forth Dimensions* for their continuing collection of informative articles.

Regards,

R.I. Demrow
P.O. Box 158 Blv. Sta.
Andover, Massachussetts 01810

Editor's reply: We do, indeed, apologize for the inconvenience which resulted from last-minute production of the mentioned pages. We are using a new production procedure which will result in gradual but noticeable improvements in our format. Reader feedback is solicited, as always.

Anonymous Variables

Leonard Morgenstern
Moraga, California

In Forth, the stack is the usual medium for passing parameters and storing intermediate results. When there are more than one or two of these, it can take a lot of programming merely to juggle the stack. Long strings of **ROT**, **SWAP**, >**R** and **R**> commands are a warning that the stack is being overused and that the situation might be better handled by means of an auxiliary variable or work area. The simplest way to make such an auxiliary is the obvious one: create a **VARIABLE** and extend it, if desired, into a work area by means of the word **ALLOT**. This has the disadvantage that dangerous conflicts can occur unless one is very careful in assigning names. Also, each such variable consumes space, not only for the data, but also for the name, link and code fields. A way to circumvent the problem is presented here using anonymous variables, which are variables that have no link or name field.

Definitions

Anonymous variables require only three definitions: **ANON/**, **ANON** and **MAKEANON**. Two other words, **ANON+** and **STORESTACK** are useful auxiliaries. These are defined in screen 100. **ANON/** is an ordinary variable.

MAKEANON creates a new anonymous variable by storing **HERE** in **ANON/** and then appending the CFA of the word **VARIABLE** to the end of the dictionary, followed by a two-byte work area with the initial value of zero. In this way a variable has been created without a name or link field. Each time **MAKEANON** is invoked, all previous anonymous variables become inaccessible. Previously compiled references are unchanged. There is a similarity to local variables in Fortran and other high-level languages. Work areas longer than two bytes may be created by following **MAKEANON** with an **ALLOT** command.

ANON is the "name" of the latest anonymous variable, which the programmer can use as if it were an ordinary variable. During compilation, the CFA of the latest anonymous variable is compiled. When not compiling, the PFA of the latest anonymous variable is put on the top of the stack.

ANON+ is used to refer to extensions of the original work area. Thus **[2] ANON+** refers to the same address as **2 ANON+** with the advantage of faster execution time and less use of memory. The brackets are necessary during compilation to prevent the value **2** from itself being compiled. **ANON+** adds the offset to the address contained in **ANON/** and, depending on the value of **STATE**, compiles the result as a literal or leaves it on the stack.

STORESTACK is a word that can be used in conjunction with **ANON** for storing parameters. Assuming that the most recent anonymous variable has been extended into a work area by means of the word **ALLOT**, **ANON** n **STORESTACK** will move the top n stack items into the work area. The value originally at the top of the stack is addressed by **ANON**, the next by **[2] ANON+**, etc. See example in screen 101.

Discussion and Examples

I create an anonymous variable every time I need a variable or work area that will be used by only one word or a small group of consecutively defined words. Words that will be needed throughout the program must be defined in the usual way. Although it can be argued that anonymous variables do nothing that cannot be done by ordinary variables, I find them very convenient and use half a dozen or so in an average program.

With ordinary variables, it is easy to inadvertently assign a name that has already been used, resulting in a hard-to-find bug (which happened to me, after which I devised the system presented here). The danger is especially severe when the program is being assembled in segments. Anonymous variables eliminate the need to search listings to determine whether a variable has been previously named. Of course, one could define an ordinary variable named **TEMP** and re-define it as many times as one wishes, but the resulting error messages are annoying and distract one's attention from real errors during compilation.

Anonymous variables entail little or no cost in execution speed and memory. The bytes used in defining the system are partly offset by the savings from the omission of the name and link fields.

There are no limitations to anonymous variables that do not also apply to ordinary variables. For example, they cannot be used in recursive situations. The stack must be used, however complex it may be.

Example One: DO LOOPs

An example of a situation in which an anonymous variable can be used to advantage is in the frequently-occurring construct,

DO . . . IF . . . LEAVE ENDIF . . . LOOP

Suppose it is desired to leave a flag on top of the stack, after exit from the loop, to indicate whether exit has occurred by exhaustion of the loop or via the **LEAVE** command. The flag will be zero if the former and one if the latter. This can be accomplished in two ways: putting a variable on the stack or using an auxiliary variable. The first is shown in figure one as method one. One puts a zero on the stack before entering the loop and rotates it so that it is below the loop limits. It is incremented if exit is via **LEAVE**. Method one increases the depth of the stack throughout the whole loop, a nuisance if the stack is already rather deep to start with. In that case, an auxiliary variable is superior (method two).

Example Two: Storing Parameters

When a word needs a lot of parameters, it is often better to move them to a work area than to leave them on the stack. Anonymous variables simplify this process. An example is given in screen 101. An eight-byte anonymous work area is created. Next, **TESTWORD** is defined, which puts four numbers on the stack, and stores them in the work area. The top of the stack goes into **ANON**, the next into **ANON+2**, etc. Finally, the second item is fetched back to the top of the stack by the sequence **[2] ANON+** @. After execution of **TESTWORD**, the value 2 is on the top of the stack.

Leonard Morgenstern has used Forth for five years and still regards himself as a student. He is a pathologist with three grown children, none of whom have any interest in computers.

Method One (Satisfactory in simple situations.)

```
0 ROT ROT
DO . . .
IF . . . 1+ LEAVE
ENDIF . . .
LOOP
```

Method Two (Superior if stack is complex.)

```
0 ANON !
DO . . . IF . . . 1 ANON ! LEAVE
ENDIF . . . LOOP ANON @
```

Figure One
Leaving flag on stack to indicate whether exit from DO loop has been via LEAVE command or by exhaustion of loop

```
  (    SCREEN 100      ANONYMOUS VARIABLES                          )

 0 VARIABLE ANON/
 : MAKEANON HERE ANON/ ! ANON/ CFA @ , 0 , ;
 : ANON ANON/ @ STATE @ IF , ELSE EXECUTE ENDIF ; IMMEDIATE

 : ANON+ ANON/ @ EXECUTE +
         STATE @ IF [COMPILE] LITERAL ENDIF ; IMMEDIATE
 : STORESTACK 2 * OVER + SWAP DO I ! 2 +LOOP ;




 (    SCREEN 101 ANONYMOUS VARIABLES EXAMPLE                        )

 MAKEANON 6 ALLOT
 : TESTWORD 4 3 2 1 ANON 4 STORESTACK [ 2 ] ANON+ @ ;
```

Interpreters *(Continued from page 19)*

High-Level Inner Interpreters

It is preferred that inner interpreters be defined in host machine code because they are expected to be called very often and it is desirable that they execute at the highest possible speed. However, machine code inner interpreters are not transportable between different CPUs, and it is not easy to program them if the interpreter gets complicated. Forth allows us to write inner interpreters in high-level code, similar to colon definitions using the **CREATE . . . DOES>** construct. We've seen two examples, **MSG** and **ARRAY**. To implement inner interpreters in high level, we have to go through another level of calling. Let's use the above two examples to illustrate how the high-level inner interpreters are implemented and executed.

In all the words defined by **MSG**, the code field contains a pointer to the machine code routine **DOMSG**. In arrays defined by **ARRAY**, the code field contains a pointer to **DOARRAY**. The code of **DOMSG** and **DOARRAY** looks like figure thirteen.

I assume in this "universal assembly code" that **CALL** first pushes the next address of the return stack, as most modern CPUs would, before jumping to the special routine **DODOES**. **DODOES** rearranges the registers so that the parameter field address of the current word using the high-level interpreter is pushed onto the data stack, **IP** is saved on the return stack, and the address after **CALL DODOES**, temporarily passed to the return stack, is copied into the **IP** register. Only then can the high-level interpreter be activated to interpret the information stored in the parameter field of the current word. While the machine-code inner interpreter gets the code field address through the **W** register, the high-level inner interpreter must get the same information through the data stack, because high-level words cannot access the **W** register directly.

Interplay Between Text and Inner Interpreters

Now that we have learned the functions of the text interpreter and of the

Continued on page 37

: CONSTANT (n ---)	
CREATE	Create an entry in the dictionary
,	and compile the constant value in the parameter field.
;CODE	End of the constant compiler and beginning of the constant interpreter.
<code of DOCON>	The constant interpreter.
: VARIABLE (---)	
CREATE	Create a dictionary entry.
2 ALLOT	Allocate parameter field.
;CODE	
<code of DOVAR>	Variable interpreter.

Figure Twelve
Creating CONSTANT and VARIABLE

DOMSG:	**W** register points to the code field of the message string.
CALL DODOES	Jump to a subroutine **DODOES** after pushing the next address on the return stack.
COUNT	Pfa is now on data stack.
TYPE	Print the string.
;	
DOARRAY:	**W** points to code field of the array word defined by **ARRAY**.
CALL DODOES	Push array base address on data stack. Start array interpreter.
SWAP	Swap array offset to top of stack.
2*	Byte to cell conversion
+	Add to array base address, pfa.
;	
DODOES:	**W** points to code field of the defined word. **RP** points to the high-level interpreter after **CALL DODOES**.
INC W	Point **W** to parameter field.
DEC SP	Make room on the data stack.
MOV W,(SP)	Push pfa of current word onto the data stack, to be used by the high-level interpreter.
MOV (RP),W	Copy the code field address of the first high-level code in the interpreter to **W** for execution.
MOV IP,(RP)	Save the **IP** pointer on the return stack, so **IP** can be used by the high-level interpreter.
MOV (W),PC	Start executing the high-level interpreter.

Figure Thirteen
Code for DOMSG and DOARRAY

Forth List Handling

Birger Olofsson
Linkoping, Sweden

A list structure is a very useful tool when building data structures of variable size, such as queues and stacks. Especially when data have to be searched or sorted, list structures give the advantage of both high speed and easy management.

The reason for this is that the stored data can be kept fixed in memory while only pointers to the data records have to be managed during moving or sorting. A list consists of one or several list elements where each element contains a pointer to the next element in the list (figure one).

The first two bytes of the list element is the link field containing a pointer to the next element. If this field is zero (null), it marks the end of the list (eol). The rest of the element is allocated for data storage.

When creating this data structure, we first have to reserve memory space for the individual list elements. Then the different elements are linked together to constitute a free list. When we need an element to operate on, we take it from the free list; and when it is no longer needed, it is returned to the free list.

Let us assume that such a list can be created by

n m CREATE-LIST<name>

where n determines the number of list elements to be created and m is the number of bytes to be allotted for data storage. The definition of **CREATE-LIST** can be found on screen #1.

CREATE-LIST is a defining word that will compile a free list <name> to the Forth dictionary. When <name> is executed it will only return its PFA to the stack. This PFA is, however, pointing at the first element in the list.

Variables also return their PFAs to the stack at execution time. This means that we can obviously use ordinary variables as lists. We simply let the value of a variable be a pointer to the first element in the list. A variable with the value zero would now represent a list which is currently empty.

The next step is to create tools for management of the individual list elements. We must be able to take an element from a list and deposit it at the top or bottom of another list. Furthermore, we must be able to keep track of the number of list elements contained in a list. This can be done with the following words, definitions for which are found in screens #2 - 3.

GET (list --- le1)

GET expects a PFA on the stack. The address of the first element (le1) is returned to the stack. The element is removed from the list.

PUT (le list ---)

PUT expects the address of a list element (le) and the PFA of a list on the stack. After execution, le is the new top element of the list.

APPEND (le list ---)

Expects a list element and PFA of a list

```
( Forth list structure   scr#1 BO 8Feb84 Forth79 )
   HEX
:  CREATE-LIST  CREATE    create a new word in the dictionary )
     HERE  2+ ,           let PFA point at link field of list )
     SWAP 1 DO             (  loop limits are n och 1 )
     HERE  OVER 2+ + ,     (  compile pointer to next element )
     DUP ALLOT             (  allot m bytes )
     LOOP                  (  repeat n-1 times )
     0 , ALLOT             (  let last element point at NIL )
     DOES>   ;
-->

(  Forth list structure scr#2 BO 8Feb84 Forth79 )

(  list --- le1 )
: GET  DUP @ DUP @         (  list le1 le2 )
             ROT !  ;      (  le1 remains )

(  le list --- )
: PUT  DUP @  3 PICK          le le1 list le )
     ! ( list le )  ! ;       stack empty )

(  le list --- )
: APPEND  OVER 0 SWAP !       let le point at NIL )
     BEGIN DUP @ WHILE        find last element )
     @ REPEAT !  ;            install le as last element )
-->

(  Forth list structure scr#3 BO 8Feb84 Forth79 )

(  list --- n )
: #LIST  0 BEGIN OVER @ WHILE 1+      (  count elements )
     SWAP @ SWAP REPEAT               (  until eol )
     SWAP DROP  ;                     (  save n on stack )

(  list --- le )
:  LAST  BEGIN DUP @ WHILE @ REPEAT

(  le list --- )
: DELIST  BEGIN DUP @ ?DUP WHILE      search for le )
     3 PICK =  IF GET 2DROP EXIT      found )
               ELSE @ THEN            next le )
          REPEAT  0 ERROR  ;          not found )
EXIT
```

on the stack. After execution, le is the last element of the list.

#LIST (list — n)

Expects the PFA of a list on top of the stack. Returns the number of list elements contained in the list.

LAST (list — le)

Expects the PFA of a list on the stack. Returns the address of the last element in the list. The list element is not removed from the list.

DELIST (le list —)

Expects a list element and the PFA of a list on the stack. Deletes le from the list. If the list element cannot be found in the list, an error message ("#0 not found") is issued.

Now we have created the necessary tools for list management. What's missing is an instrument that facilitates positioning within a list element. This can, of course, easily be done by adding an offset to the start address of the list element. By designing another defining word this can be done in a manner resembling the record structure of Pascal.

RECORD

<name> (n —) compiling
<name> (addr — addr+n) executing

At compile time, <name> is created in the dictionary and the number n on the stack is compiled into its PFA. At execution time, the offset n is added to the address on the stack. The definition is:

```
: RECORD  CREATE
  , DOES > @ + ;
```

We finish our discussion by giving a few examples of how the list structure can be used. If, for some application, we need twenty list elements where each element is supposed to contain sixty-four bytes of information, we write:

20 64 CREATE-LIST FREELIST

We also need a list **SUB1** for data storage. This list is created as an ordinary variable.

VARIABLE SUB1 0 SUB1 ! (list is empty)
FREELIST GET SUB1 PUT
FREELIST GET SUB1 APPEND

We now have taken two elements from the free list and put them in the same order in the sub-list. The sub-list now contains two elements:

SUB1 #LIST .
2 ok

To get to the data field of the element we may create a data pointer.

2 RECORD DATA (create pointer to data field)

The use of the list structure is now, perhaps, obvious. The only important thing is to keep track of the elements so that they are not lost. After processing, the elements must be returned to the free list.

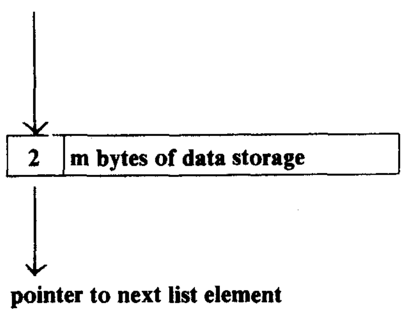


Figure One

Interpreters *(Continued from page 35)*

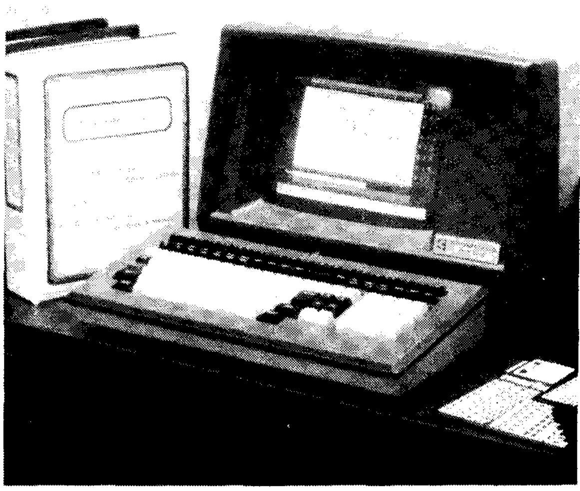
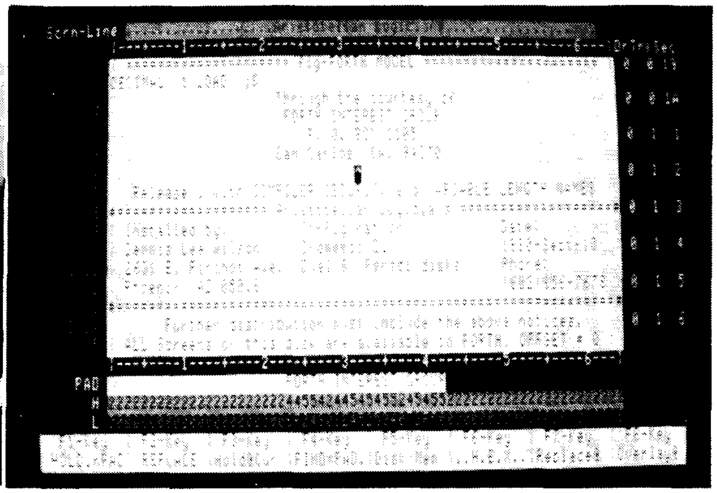
many inner interpreters, a question remains: What's the significance of separating the functions of the inner interpreters from those of the text interpreter?"

Before we answer this question, let us digress for the moment to what a typical interpreter, like that of BASIC, has to do to get its job done. When a line of commands or code is typed into the terminal, the BASIC interpreter must first subject this line of commands to a syntax analysis (according to a set of rather complicated syntax rules) to determine what actions will take place. It must determine whether a line number is present or not. Without a line number, it will expect direct commands; otherwise, it will translate the line to a form suitable for internal storage. It has to separate all the keywords in the line because the keywords represent functions built in the BASIC system. It has to assign values to variables and evaluate expressions according to rules associates with the keywords detected, etc., etc.

The compiled languages, like Fortran, are even worse. The compiler has to detect all the keywords, assigned or unassigned variables, analyze syntax, determine functions, etc., before it can generate any code. The complexity in syntax and semantics in the Fortran language makes it impossible for us mortal souls to understand the contents of a Fortran compiler, not to mention changing it.

How, then, can the text interpreter in Forth be so simple?

The reason is that the Forth text interpreter does not have to do any syntax analysis of the lines of commands given to it, because there is no syntax to be analyzed! All the text interpreter has to do is parse out words from the command line and search the dictionary to find the commands. It does not have to know anything about the commands, whether they are constants, variables, colon words, or code words. It just has to find the word in the dictionary and turn the code-filled address over to the inner interpreter pointed to by the contents of

8080/Z80 FIG-FORTH for CP/M & CDOS systems
FULL-SCREEN EDITOR for DISK & MEMORY

$50 saves you keying the FIG FORTH model and many published FIG FORTH screens onto diskette and debugging them. You receive TWO diskettes (see below for formats available). The first disk is readable by Digital Research CP/M or Cromemco CDOS and contains 8080 source I keyed from the published listings of the FORTH INTEREST GROUP (FIG) plus a translated, enhanced version in ZILOG Z80 mnemonics. This disk also contains executable FORTH.COM files for Z80 & 8080 processors and a special one for Cromemco 3102 terminals.

The 2nd disk contains FORTH readable screens including an extensive FULL-SCREEN EDITOR FOR DISK & MEMORY. This editor is a powerful FORTH software development tool featuring detailed terminal profile descriptions with full cursor function, full and partial LINE-HOLD LINE-REPLACE and LINE-OVERLAY functions plus line insert/delete, character insert/delete, HEX character display/update and drive-track-sector display. The EDITOR may also be used to VIEW AND MODIFY MEMORY (a feature not available on any other full screen editor we know of.) This disk also has formatted memory and I/O port dump words and many items published in FORTH DIMENSIONS, including a FORTH TRACE utility, a model data base handler, an 8080 ASSEMBLER and a recursive decompiler.

The disks are packaged in a ring binder along with a complete listing of the FULL-SCREEN EDITOR and a copy of the FIG-FORTH INSTALLATION MANUAL (*the language model of FIG-FORTH, a complete glossary,* memory map, installation instructions and the FIG line editor listing and instructions).

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

	USA	Foreign AIR
☐ **FIG-FORTH & Full Screen EDITOR package**		
Minimum system requirements:		
80x24 video screen w/ cursor addressability		
8080 or Z80 or compatible cpu		
CP/M or compatible operating system w/ 32K or more user RAM		
Select disk format below, (soft sectored only)	$50	$65

☐ 8″ SSSD for CP/M (Single Side, Single Density)

Cromemco CDOS formats, Single Side, S/D Density

☐ 8″ SSSD ☐ 8″ SSDD ☐ 5¼″ SSSD ☐ 5¼″ SSDD

Cromemco CDOS formats, Double Side, S/D Density

☐ 8″ DSSD ☐ 8″ DSDD ☐ 5¼″ DSSD ☐ 5¼″ DSDD

Other formats are being considered, tell us your needs.

	USA	Foreign AIR
☐ Printed Z80 Assembly listing w/ xref (Zilog mnemonics)	$15	$18
☐ Printed 8080 Assembly listing	$15	$18
TOTAL	$______	______

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

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

Chapter News

John D. Hall
Oakland, California

We have four new chapters. That makes fifty!

Tucson FIG chapter
Tucson, Arizona

Southeast Florida FIG Chapter
Miami, Florida

Central Illinois FIG Chapter
Urbana, Illinois

Berkeley FIG Chapter
Berkeley, California

Orange County FIG Chapter

December 7, Wil Baden conducted a contest on who could write a definition of DIGIT in the fewest number of words. (Okay guys, what is it?) Wil then reported on his trip to FORML. January 4, Jim Flournoy spoke about his trip to Sweden and his meeting with members of the Forth community. Noshir Jesung spoke about a System Index which he and Wil Baden implemented in the F83 model. January 25, Art Horne demonstrated his Rockwell single-board development system. The system included four disk drives, CRT and keyboard. He sells the boards in small volume for Rockwell. Wil Baden presented a simple file system for the F83 model. Elections were held and the results were Noshir Jesung re-elected President, Bob Wada as Vice-President, Roland Koluvek as Secretary and Bob Snook as Treasurer. February 1, D.E. Legan presented a paper on VIC modem screens. He has been using his VIC to communicate with an IBM mainframe.

Taipei FIG Chapter Activities

September 24, 1983

13:00 Forth Fundamentals I, Mr. Chin-tan Cheng
14:00 Digital Filter Design, Mr. Ming-sun O-yang
15:00 6522 Interrupts, D/A Converter, Mr. Yu-chu Lin
16:00 FIG business meeting

October 22, 1983

13:00 Forth Fundamentals II, Mr. Son-tang Shiu
14:00 DBMS Concepts, Mr. Rei-se Chen
15:00 AIM-65 With a Four-Channel Gamma Spectrometer, Mr. Ke Hwang
16:00 FIG business meeting

November 26, 1983

13:30 Forth Fundamentals III, Mr. Song-li Chu

---

the code field. The inner interpreters will take care of the rest, carry out all the work designed into the word definitions.

The inner interpreters form an insulating layer, hiding the complexities of a real computer under it and presenting to the text interpreter, and to the final users, a simple, uniform and yet very powerful interface. Not only can the user use the inner interpreters provided in a regular Forth system to define new commands, he is also provided with all the tools to build new compiler/interpreters which can be used to build specialized commands and data structures best suited for his own applications.

Code fields and the associated inner interpreters are the sole inventions Mr. Charles Moore brought us in Forth. Stacks, the dictionary, indirect threaded code and virtual memory were all well-developed techniques before Forth was invented. Using the code field to identify a specific interpreter to execute a particular command was not obvious or considered useful prior to that time. The code field sets Forth apart from any other type of language or programming constructs, and it is the most unique feature in Forth or Forth-like systems. Many of the attributes associated with the Forth language, such as compactness, simplicity and extensibility, can only be realized with the use of the code field.

This concept of compiler/interpreter pairs very neatly ties up many loose ends in the understanding of the Forth computer, such as the code fields, the nested execution of Forth words and the very confusing idea of defining words. This article presents my personal view on this many-spotted Forth beast. I hope this discussion can shed some light for the newcomers to this language.

References

1. L. Baker, M. Derick, *Pocket Guide to Forth*, Addison-Wesley, 1983, p.4.
2. K. Harris, "Forth Extensibility", *Byte*, Vol. 5, No. 8, 1980, p. 164
3. L. Brodie, *Starting Forth*, Prentice-Hall, 1981, p. 215.

14:30 Bit-Slice Microprocessors, Mr. Chi-yi Liu
15:30 Micromotion Apple Forth-79 Internals, Mr. Sam Chen
16:30 FIG business meeting

December 24, 1983

13:30 Forth Fundamentals IV: Dictionary, Vocabulary and Definitions, Mr. Ching-lun Lee
14:30 Digital Communications, Mr. Yu-sen Tzai
15:30 A/D Converter Applications, Mr. Sing-tan Cheng
16:30 FIG business meeting

January 28, 1984

13:30 Forth Fundamentals V: User Variables, Mr. Chi-liu Kan
14:30 Data Transfer Between Apple IIs, Mr. Yi-seu Wei
15:30 Forth for a TTL IC Tester, Mr. Sui-shan Lan
16:30 FIG business meeting

Kansas City FIG Chapter

February 23, nine people attended the last meeting. Les Lovesee gave an excellent demonstration of a meta-compiler. This prompted discussion about Forth and it was decided to continue an open forum on Forth at the next meeting. The group received their first order from Mountain View Press. It was a complete success. A new order has been started.

Syracuse FIG Chapter

December 14, 1) Introduction of attendees. 2) Established a permanent meeting time (3rd Wednesday, 7:30 p.m.), place varies. 3) Gave out phone numbers for local public access bulletin boards. 4) Made a list of topics to be covered in future meetings. 5) Adjourned for casual use of Forth on two systems brought to the meeting by Mark Manning and Bill Carlson.

January 18, the group developed a meeting format to be followed in the future. It will be a two-hour meeting with a short business meeting, half-hour tutorial, half-hour special topic and then a new set of hardware demos of Forth. Elections will be held in February.

February 15th minutes of the Syracuse FIG Chapter:

1. The minutes of 1/18/84 were read and approved.
2. Treasurer reported $50 in bank and another $15 in dues collected.
3. It was decided to elect the officers until September 1984 and every September thereafter, and that the initial slate should have a President, Vice-President, Secretary/Treasurer, Program Chairman, Public Relations/Membership Chairman and a Demo Chairman.
4. The following were elected:
President: John DeMar, 456-2237 days
Vice-President: Brad McLean, 437-1375
Secty./Treas.: Dick Corner, 456-7436 days
Prog. Chair.: Hank Fay, 446-4600
Demo. Chair.: Dick Sutliff, 478-0931
Pub. Rel./Memb. Chair.: Alan Rowoth, 474-4800
5. Held tutorial on Chapter One of Brodie by Dick Corner.
6. Hardware demo of C64 by Hank Fay.
7. CompuServe demo by John DeMar.

John D. Hall is the National Chapter Coordinator for the Forth Interest Group.

New Chapters in Formation

Here are more of the new chapters that are forming. If you live in any of these areas, contact these people and offer your support and help in forming a FIG chapter. You are not expected to be one of the Forth "experts"; the job of organizing a chapter can be done as well by people who are better at organizing than at programming, or by people who are in need of the help and support that a chapter can return. Lend a hand!

Ron Braithwaite
2364 Lochridge Pl.
Escondido, CA 92026
619/745-1089

John Haddock
580 Island View Circle
Port Hueneme, CA 93041

Mathew Ferens
2201 West Thomas
Hammond, LA 70401

Marc L. Bellario
Interplanetary Software
427 Plum Ave.
Lakewood, MN 55804

George Y. Lee, Jr.
1118 Archdale Dr.
Charlotte, NC 28210

Eli Camhi
Sesame Enterprises
2101 Myrtle Ave.
El Paso, TX 79901
915/582-2816

Claude W. Hesselman
2545 Bainbridge Blvd.
Chesapeake, VA 23324
804/545-1240

Jean-Marc Bertinchamps
Rue N. Monnom, 2
B-6290 Nalinnes
Belgium

Basil Barnes
10348 - 146th St.
Edmonton, Alberta T5N 3A2
Canada

John Clark Smith
363 Woodfield Rd.
Toronto, Ontario M4L 2W9
Canada

Erik Ostergaard
COMPEX
2 Gertsvej
2300 Copenhagen S.
Denmark

Klaus & Angelika Flesch
Vertrieb Von Software
Schuetzenstrasse 3
7820 Titisee-Neustadt
West Germany

David K. Walker
Altakonsult
Postboks 68
Langeesen 16, 5084 Tertnes
Norway

FIG Chapters

U.S.

• ARIZONA

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

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

• CALIFORNIA

Berkeley Chapter
Monthly, 2nd Sat., 1 p.m.
10 Evans Hall
University of California
Berkeley
Call Mike Perry
415/624-3421

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

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

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

Silicon Valley Chapter
Monthly, 4th Sat., 1 p.m.
Dysan Auditorium
5201 Patrick Henry Dr.
Santa Clara
Call Glenn Tenney
415/524-3420

• COLORADO

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

• FLORIDA

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

• ILLINOIS

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

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

Rockwell Chicago Chapter
Call Gerard Kusiolek
312/885-8092

• INDIANA

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

• IOWA

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

• KANSAS

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

• MASSACHUSETTS

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

• MINNESOTA

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

• MISSOURI

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

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

• NEVADA

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

• NEW YORK

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

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

Syracuse Chapter
Monthly, 3rd Wed., 7:30 p.m.
Call C. Richard Corner
315/456-7436

• OHIO

Athens Chapter
Call Isreal Urieli
614/594-3731

Cleveland Chapter
Call Gary Bergstrom
216/247-2492

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

• OKLAHOMA

Tulsa Chapter
Monthly, 3rd Tues., 7:30 p.m.
The Computer Store
4343 South Peoria
Tulsa, OK
Call Art Gorski
918/743-0113

• OREGON

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

• PENNSYLVANIA

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

• TEXAS

Dallas/Ft. Worth
Metroplex Chapter
Monthly, 4th Thurs., 7 p.m.
Software Automation, Inc.
14333 Porton, Dallas
Call Chuck Durrett
214/788-1655
Bill Drissel
214/788-1655
Bill Drissel
214/264-9680

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

• VERMONT

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

• VIRGINIA

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

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

FOREIGN

• AUSTRALIA

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

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

• BELGIUM

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

• CANADA

Nova Scotia Chapter
Contact: Howard Harawitz
227 Ridge Valley Rd.
Halifax, Nova Scotia B3P 2E5
902/542-7812

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

• COLOMBIA

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

• ENGLAND

Forth Interest Group — U.K.
Monthly, 1st Thurs., 7 p.m.
Bradden Old Rectory
Towchester, Northamptonshire
NN12 8ED
Contact: Keith Goldie-Morrison
15 St. Albans Mansion
Kensington Court Place
London W8 5QH

• FRANCE

French Language Chapter
Contact: Jean-Daniel Dodin
77 rue du Cagire
31100 Toulouse
(16-61) 44.03.06

• IRELAND

Irish Chapter
Contact: Hugh Dobbs
Newton School
Waterford
051/75757
051/74124

• ITALY

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

• SWITZERLAND

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

• REPUBLIC OF CHINA

R.O.C.
Contact: J.N. Tsou
Forth Information Technology
P.O. Box 53-200
Taipei
02/331-1316

SPECIAL GROUPS

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

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

Detroit Atari Chapter
Monthly, 4th Wed.
Call Tom Chrapkiewicz
313/524-2100

FIGGRAPH
Call Howard Pearlmutter
408/425-8700

FORTH INTEREST GROUP

MAIL ORDER

	USA	FOREIGN AIR
☐ Membership in FORTH Interest Group and Volume V of FORTH DIMENSIONS	$15	$27
☐ Back Volumes of FORTH DIMENSIONS. **Price per each.** ☐ I ☐ II ☐ III ☐ IV	$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 Listings of fig-FORTH for specific CPU's and machines. The above manual is required for installation. Check appropriate box(es). **Price per each.**	$15	$18

☐ 1802 ☐ 6502 ☐ 6800 ☐ 6809 ☐ VAX ☐ Z80
☐ 8080 ☐ 8086/8088 ☐ 9900 ☐ APPLE II ☐ ECLIPSE ☐ IBM PC
☐ PACE ☐ NOVA ☐ PDP-11 ☐ 68000 ☐ ALPHA MICRO

	USA	FOREIGN AIR
☐ "Starting FORTH, by Brodie. BEST book on FORTH. (Paperback)	$18	$22
☐ "Starting FORTH" by Brodie. (Hard Cover)	$23	$28

☐ PROCEEDINGS: FORML (FORTH Modification Conference)
- ☐ 1980, $25USA/$35Foreign
- ☐ 1981, Two Vol., $40USA/$55Foreign
- ☐ 1982, $25USA/$35Foreign

ROCHESTER FORTH Conference
- ☐ 1981, $25USA/$35Foreign
- ☐ 1982, $25USA/$35Foreign
- ☐ 1983, $25USA/$35Foreign — Total $________

☐ STANDARD: ☐ FORTH-79, ☐ FORTH-83. $15USA/$18Foreign EACH. — Total $________

☐ Kitt Peak Primer, by Stevens. — $25 — $35

☐ MAGAZINES ABOUT FORTH: ☐ BYTE Reprints 8/80-4/81
☐ Dr Dobb's Jrnl, ☐ 9/81, ☐ 9/82, ☐ 9/83
☐ Poplar Computing, 9/83 $3.50USA/$5Foreign EACH. — Total $________

☐ FIG T-shirts: ☐ Small ☐ Medium ☐ Large ☐ X-Large — $10 — $12

☐ Poster, BYTE Cover 8/80, 16"x22" — $ 3 — $ 5

☐ FORTH Programmer's Reference Card. If ordered separately, send a stamped, self addressed envelope. — Free

TOTAL $________

NAME________________________________ MS/APT____________

ORGANIZATION________________________ PHONE(   )____________

ADDRESS____

CITY____________________ STATE______ ZIP__________ COUNTRY__________

VISA#____________________ MASTERCARD#____________________

Card Expiration Date____________

(Minimum of $15.00 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. 10/83

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. 3107
San Jose, CA

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

Address Correction Requested