Volume 7, Number 1

FORTH Dimensions

May/June 1985
$2.50

Forth Spreadsheet

Not ONLY But ALSO

Menus in Forth

Macro Generation

Lots o' Variables

FORTH Dimensions
Published by the
Forth Interest Group

Volume VII, Number 1
May/June 1985

Editor
Marlin Ouverson

Production
Cynthia Lawson

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 to 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 to submit material for publication, the address is: Forth Interest Group, P.O. Box 8231, San Jose, California 95155.

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.

Code and examples conform to Forth-83 standard.

Code and examples conform to Forth-79 standard.

Code and examples conform to fig-FORTH.

Deals with new proposals and modifications to standard Forth systems.

FORTH Dimensions

FEATURES

DEPARTMENTS

Now You Can Add

ARTIFICIAL INTELLIGENCE

New!

To Your Programs Using a Powerful Combination

"SUPERFORTH 64" +AI

By Elliot Schneider & Jack Park

Heres Your Chance to Profit by being on the Forefront, Write 5th Generation Software

Learn How To:

- Create Intelligent Programs
- Build Expert Systems
- Write Stand Alone License Free Programs
- Construct Rule Bases
- Do Knowledge Engineering
- Use Inference Engines

Write Intelligent Programs For:

- Home Use
- Robotics
- Medical Diagnosis
- Education
- Intelligent CAI
- Scientific Analysis
- Data Acquisition
- Data Analysis
- Business
- Real Time Process Control
- Fast Games
- Graphics
- Financial Decisions

Extended Math Functions

- Fast ML Floating Point & Integer Math
- Double Precision 2E+38 with Auto. Sci Not.
- $n^x e^x$ Logx Loge Sin Cos Tan SQR 1/X...
- Matrix and Multidimensional Lattice Math
- Algebraic Expression Evaluator

Easy Graphics & Sound Words

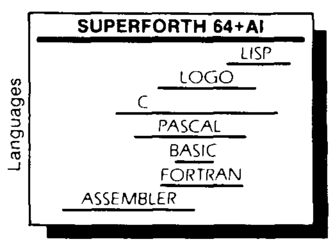
- Hires Plotting
- Windows
- Split Screen
- Printer/Plotter Ctrl
- Sprite & Animation Editor
- Turtle Graphics
- Koala Pad Graphics Integrator
- Hires Circle, Line, Arc
- Music Editor
- Sound Control

Easy Control of all I/O...

- RS232 Functions
- Access all C-64 Peripherals
- Interrupt Routines

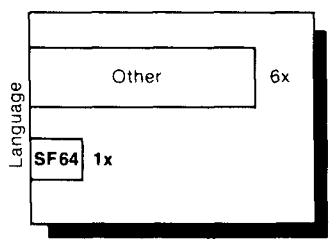
Utilities

- Interactive Interpreter
- Forth Virtual Memory
- Full Cursor Screen Editor
- Full String Handling
- Trace & Decompiler
- Conditional Macro Assembler
- Interactive Compiler
- Romable Code Generator
- 40K User Memory
- All Commodore File Types
- Conversational User Defined Commands



Power of Languages Constructs

SuperForth 64 is more powerful than most other computer languages



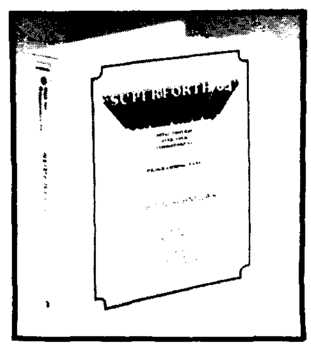
Programming Time

SuperForth 64 Saves You Time and Money

Great Documentation

- Easy to Read 350 pg. Manual with Tutorials
- Source Screen Provided
- Meets all MVP Forth-79 Industrial Standards
- Personal User Support

A Total Integrated Package for the Commodore 64



Ordering Information: Check, Money Order (payable to MOUNTAIN VIEW PRESS, INC.), VISA, MasterCard, American Express. COD's $5.00 extra. No billing or unpaid PO's. California residents add sales tax. Shipping costs in US included in price. Foreign orders, pay in US funds on US bank, include for handling and shipping $10.

* Parsec Research
Commodore 64 TM of Commodore

SPECIAL INTRODUCTORY OFFER

only $99.00

$203.00 Value
Limited Time Offer

Call:
(415) 961-4103
MOUNTAIN VIEW PRESS INC
P.O. Box 4656
Mt. View, CA 94040

Dealer for
PARSEC RESEARCH
Drawer 1776, Fremont, CA 94538

Questions Standard Procedure

Dear Sir:

Since publication of my Forth-83 article in *BYTE* (August 1984), I have received dozens of letters. Most readers seem to agree with Nicholas Pappas' letter in *Forth Dimensions* (VI/5) that decried the continuing growth of new "standard" dialects.

I took no overt position in my article — it was a report, not an editorial — but I must agree. Forth-83 is marginally better than fig-FORTH or Forth-79 in some respects, but the changes do not significantly increase the language's power. Moreover, the changes are often just as subtle as they are radical. I feel sorry for the novice struggling to learn the differences in such fundamental items as division and do loops.

Some of the most interesting correspondence resulting from the *BYTE* article has been reprint requests, almost all from government or university officials of Soviet-bloc nations: Poland, Cuba, East Germany, U.S.S.R., etc. In each case, I have forwarded copies of the article along with a request for information on Forth use in their nation. I am enclosing one interesting response from Warsaw, Poland.

Thanks for your help.

Very truly yours,

C. Kevin McCabe
Chicago, Illinois

Simpler Recursion

Dear Editor:

In the letter on recursion (*Forth Dimensions* VI/5), the suggestion can be made even simpler. Make **SMUDGE** immediate (if it isn't already) by [ ]. Then, in Forth-83, **GCD** becomes:

```
: GCD [SMUDGE]
?DUP IF  SWAP  OVER  MOD GCD
[SMUDGE]  THEN  ;
```

Peter Oppenheimer
Princeton, New Jersey

More Grass Roots

Dear FIG,

I just read in *Forth Dimensions* (VI/5) a letter by Lionel Hewett, which you entitled "Grass-Roots Forth." I had to reread the name several times to make sure that *I* had not written that letter. I could have... word for word.

The article "How to Learn Forth" was the first article I have read in the five issues I have received of *Forth Dimensions* that was useful to me, a beginner in Forth. It informed me through the evaluation that *both* of the Forth implementations I have bought (at over $30 apiece) are doing my attempts at learning Forth more harm than good.

Other purchases and investigations that have been useless in my attempts at learning Forth are the *6502 Source Listing* and *fig-FORTH Installation Manual*. Both at $15 and both from you.

Not counting the three books that were poor at best, I have over $100 in Forth material I can't use. (I have both of Brodie's books and they are *good*.)

My point is: I am *very* interested in learning Forth, but everywhere I turn, I'm putting out cash and getting no where. Lionel said it best in his letter. Why can't I get a good, cheap implementation of Forth for my specific machine?

Soon, I will be upgrading my VIC-20 to a Commodore 64. I have no plans to attempt Forth on my new machine unless I see some changes in the Forth community to be more "user friendly" (did I really say that?) to us beginners.

It will make me unhappy to abandon this otherwise exciting project.

Enclosed is one more renewal of my membership in FIG, in hopes that things will change. I hope it won't be my last.

Sincerely,

J. Grant Viening
Wyoming, Michigan

Thanks; it's the noisy disk drive that gets the most attention, so please tell us how we are and aren't serving your needs, as this reader has done. While we can't require vendors to adhere to the Forth standards or to publish more complete tutorials and documentation, we can try to help you over the largest obstacles, if you let us know about your problem spots. Write to "Ask the Doctor" with specific questions!

—Editor

Capital Idea

Dear Marlin,

I thoroughly agree with the comments by Jeffrey Lotspiech and Thomas Ruehle ("Automatic Capitalization in Forth," *Forth Dimensions* VI/1) regarding the superior readability of lower-case Forth words. (Under their scheme, lower case may be used if desired for newly-defined words, while upper case is retained for the standard Forth words. All text may be typed in lower case, and is automatically capitalized where necessary.)

I would like to continue discussion in this area and question why we need to keep using exclusively upper case for the standard Forth words. Many Forths already allow case to be ignored

(cont. on p. 7)

Forth Interest Group: An International Service Organization

Forth Dimensions begins Volume VII this month, initiating another year of outstanding international service and activities by the Forth Interest Group. Let's take a moment to look at the past year and at some of the plans for this year.

Growth continued, and many new FIG Chapters were added to the roster. Forth Interest Group members have organized Chapters world wide, which demonstrates the international interest in Forth. One of the largest and most active Chapters is the Republic of China's Association of ROC Forth Language. This group hosted a three-day international FORML conference at Taiwan's Tam-Kang University in September. Attendance exceeded 100, with several U.S.A. Forth Interest Group members attending and presenting papers. One paper presented Forth programmed in Chinese, to demonstrate the versatility of Forth.

Our first trip to China to participate in FORML conference programs was completed. It included a two-day conference at Shanghai's Jiao Tong University and additional university programs in Peking and Xian. We learned that China is eager to use Forth and has instituted programs in the universities so that students may learn and practice Forth. We also learned that China welcomes visitors and will keep one busy from morning till night visiting cultural centers, historical sites, factories, shopping centers, restaurants, etc.

In the U.S.A., the Forth Interest Group's annual two-day convention was held in October in Palo Alto, California. Vendors exhibited an impressive array of Forth products. Technical sessions were excellent and included hands-on training for anyone interested in learning Forth.

The FORML Asilomar Conference in November had nearly 100 participants, with a wide range of papers presented. Here was an opportunity to meet with top-flight Forth practitioners. Charles Moore, inventor of Forth, listed the remarkable capabilities of his Forth "chip," then in the final stages of development. Today, working chips are available and the promises of November are a reality.

New books about Forth were published in the past year, including *Thinking Forth*, *Mastering Forth* and *Forth Tools*. These are excellent books and are available along with others from the Forth Interest Group. Each issue of *Forth Dimensions* has a publication order form.

This year, the Forth Interest Group has already presented continuous one-hour training sessions over three days of the West Coast Computer Faire in San Francisco. Apple and IBM computers were available for individual use. This was a very popular event.

In September of 1985, the annual FIG convention is scheduled in Palo Alto, California. A complete conference program is planned to include the latest software and hardware developments. Look forward to hardware developments based on the new Forth chip. Training will continue to be an important part of the technical program.

A European conference is planned in October in Germany. It is called euroFORML and will be held in Stettenfels Castle near Heilbronn. This continues the international conference programs which have always been a part of the Forth Interest Group's activities.

You will continue to find new publications listed in the publications order form. The publications committee reviews and recommends publications regularly for this list. The Forth Interest Group believes that the publication service is very important in making publications available for world-wide distribution.

Forth Dimensions articles are a constant source of new and educational material about Forth. You are encouraged to recommend it to everyone interested in learning more about Forth and about the benefits of its use.

These are activities the Forth Interest Group supports in meeting its goals and objectives of service to members and promotion of Forth. Your support is necessary to keep these services available. Participate in Forth Interest Group events and tell others about them.

—Robert Reiling
President, Forth Interest Group

(cont. letters)

in dictionary searches, and this would permit all-lower-case Forth or, perhaps more usefully, Forth in which upper case may be used selectively to highlight whatever we want. For some time now, I have been writing code in which upper case is used for each word as it is defined, and otherwise everything is in lower case. The result looks unconventional, but it is very readable once you get used to it. (I challenge readers to try it!)

Why, then, do we persist with upper-case Forth? The only reason I can think of is tradition. Early keypunches, printers, etc. had only one case (upper), so languages such as Fortran and COBOL used upper case only. Those of us old enough to remember that Pascal had a forerunner called Algol will realize that it was an exception; but one of its intended purposes, perhaps the primary one, was the publication of algorithms, not simply the programming of computers. That is, it was intended for people to read. I think the point is obvious.

One possible objection that supporters of upper case might raise is that upper-case code stands out clearly from lower-case comments. I believe, however, that comments can be separated out just as well by moving them over to the right, onto separate lines or, even better, to shadow screens.

Most programmers today are used to the "lower-case look" of Pascal and C, both of which followed the Algol style of appearance. Writing Forth in upper case makes our program code more reminiscent of Fortran, COBOL or even (gasp, horror) BASIC! We are entering an era of bit-mapped displays and smart printers capable of handling all kinds of esoteric scripts. Should we persist with program text that looks like something out of the 1950s? I know nostalgia has its place, but surely this isn't it. Programs need to be read by people as well as by machines. If I'd written this letter in all upper case, everyone would have thought I was being ridiculous!

Yours sincerely,

Michael Hore
Numbulwar, NT, Australia

Evaluation

William F. Ragsdale
Hayward, California

"Ask the Doctor" is Forth Dimensions' *health maintenance organization devoted to your aid in understanding and using Forth. Questions of a problem-solving nature, on locating references, or just regarding contemporary techniques are most appropriate. When needed, your good doctor will call in specialists. Published letters will receive a preprint of the column as a direct reply.*

In his last two columns, the doctor addressed two approaches to learning Forth. First (*Forth Dimensions* VI/5) was a study-guide approach to learning from Leo Brodie's *Starting Forth*. Next we made rounds within the clinic (VI/6) to review Margaret Armstrong's *Learning Forth*. In this issue, we conclude by summarizing the evaluations, contributed by readers of *Forth Dimensions*, of commercial Forth systems.

Your report from the clinic for this issue has been built upon the contributions of eleven readers. Appreciation is in order for the efforts of Jim Henderson (Thomson, Georgia), Chris McCormack (Huber Heights, Ohio), Guy Kelly (La Jolla, California), Terry Jaco (North Hollywood, California) and J.C. Halbrook (Sterling, Connecticut). Several others supplied evaluations but did not identify themselves.

Summary

Previously in *Forth Dimensions*, several reader's questions regarding learning Forth were summarized by the good doctor:

- How can I get started? - Which Forth? - Whom do I ask?

The "Which Forth" question will be addressed by reporting upon the results of the questionnaire that concluded that column. The scoring method favored use of a standardized dialect, consistency with *Starting Forth*, documentation and support. It was suggested that a point total of seven or greater would indicate a system offering superior value to anyone learning Forth. The implication: a score of six or less indicates a system which will impede your learning effort.

The curtain is about to be raised. The audience is waiting with hushed expectation. The evaluations are in! May I have the sealed envelope, please?

Summary

We see from table one that the point total ranges from three to twelve. The maximum possible was thirteen. As mentioned, the scoring favors systems matching an established standard (Forth-79 or Forth-83) and the book *Starting Forth*. Both of these elements are supportive of self study.

Our mail continues to confirm that systems weak on documentation and standardization are most associated with plaintive calls for aid. These are mostly fig-FORTH systems in public-domain libraries. Our conclusion is that the $50 to $150 saved over a commercial product will be quickly offset in the added frustration and extra effort of learning Forth and the specifics of the implementation.

Three readers evaluated SuperForth 64 for the Commodore 64. All three emphatically praised the support and helpful attitude of Parsec Research. The 250-page manual, access to host files, decompiler/trace option and floating point are all given high marks. One quote: "I've dealt with Parsec for almost a year and have had very great success with their product and with their personnel. As a learning tool, I would find it hard to match the price/performance of a C-64 running SuperForth."

Two readers evaluated C64 Forth from Performance Micro Products. With a score of 10, it only lost points for the editor, which is tailored to match the Commodore conventions rather than the usual Forth keys. The dialect is Forth-79 enhanced by a file interface, 167 pages of documentation, graphics and trace.

One reader extended the rating scale to favor his choice, MMS Forth from Miller Microcomputer Services. This evaluator bumped MMS Forth to 31 points, since raves were given to the editor, flexible use of RAM, and options. That survey form's point for support was inflated to three due to the excellent phone help. In fairness to all, on the uniform scale, this system was a twelve on the scale of thirteen possible points.

NGS Forth, 8086 Forth (LMI) and 83 Standard PC-Forth (Kelly) are all available for the IBM PC. They all scored twelve, and any should be well received by the student.

F83 is a public-domain system developed by Mike Perry and Henry Laxen for the IBM PC, CP/M and 68000 systems. While technically outstanding, it only lost points for no support and lack of printed documentation. Addition of Dr. Ting's *Inside F83* (280 pages, published by Offete Enterprises) raises this system to twelve points.

The only problem case reported was VIC Forth for the Commodore VIC-20. This fig-FORTH based system only got points for object size and editor. The dialect, mass storage, support and options received no points. The manufacturer has gone out of business, but the product is still in distribution. This style system is being displaced in the market and illustrates the difficulty a newcomer may inadvertently face.

If you perform your own evaluation or select a product based on this evaluation, please remember that its purpose has been to indicate suitability for learning, and that seven or better is recommended. Other ratings would be appropriate for purposes such as product implementation or specific applications.

Other Systems

Several popular systems are noticeable by their absence. Evaluations of such systems as MVP FORTH, polyFORTH II, MasterFORTH and MacForth would be appreciated. Your faithful practitioner will also welcome further comments and evaluations that

PRODUCT	VENDOR	RUNS ON	PRICE	POINTS
8086 Forth	Laboratory Microsystems	CP/M-86	$100	12
MMS Forth	Miller Microcomputer	TRS-80	130	12
		IBM PC	250	
NGS Forth	Next Generation Software	IBM PC	70	12
83 PC-Forth	G. M. Kelly	IBM PC	25	12
F83	No Visible Support	IBM PC	25	11
		CP/M, 68000		
SuperForth 64	Parsec Research	Comm.-64	96	10
C64 Forth	Performance Microproducts	Comm.-64	70	10
VIC Forth	HES	VIC 20	40	3

Table One

may be summarized in a final tabulation for *Forth Dimensions*.

When next we summarize reader evaluations, you will find the good doctor trading his white lab coat for formal dinner attire. We will never be as glamorous as the Academy Awards, but the appreciation of readers will be more sincere.

Vendor Addresses

Laboratory Microsystems, Inc., P.O. Box 10430, Marina del Rey, CA 90295, (213) 306-7412.

G. M. Kelly, 2507 Caminito La Paz, La Jolla, CA, 92037.

HES (out of business), product dist. by Mountain View Press, P.O. Box 4656, Mt. View, CA, 94040, (415) 961-4103.

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

Next Generation Systems, P.O. Box 2987, Santa Cruz, CA 95055.

No Visible Support Software, Box 1344, 2000 Center Street, Berkeley, CA 94074.

Parsec Research, Drawer 1776, Fremont, CA 94538.

Performance MicroProducts, P.O. Box 370, Canton, MA, 02120, (617) 828-1209.

About the author

Bill Ragsdale has been using Forth since 1977 for personal and business projects. He is married to Anne, who did the production work on early *Forth Dimensions*. They have two children: Mary, age three and Michael, age one. For those of you who have been following Mary's development, she now knows the alphabet and enjoys "Kiri's Hodge-Podge" on the Apple II (which she calls E-I-Oh, as in Old McDonald) and "My ABCs" on the PC. Michael's computer involvement is limited to chewing on its mouse-control wire.

Application Tutorial

Generic Sort

John S. James
Santa Cruz, California

Application Tutorials focus on using Forth to get results, not on experimental developments. This article advocates a design approach which employs the strengths of Forth to help write generic library routines, which can be used with no change at all in different applications.

As an example, we present a simple routine to sort any kind of randomly-accessible data, in memory or on disk: numbers, records of any length, records with one or more key fields and with ascending, descending or mixed sequences, variable-length records, arrays or other data structures, or mathematical entities with any "order" relationship, not necessarily alphabetical or numerical. You can sort any of this data with no change at all to the sort routine. So you don't need to read or understand the sort in order to use it!

The essence of what we call "generic" design is the radical separation between an algorithm and its data. We use the well-known technique of vectored execution — allowing one routine to accept a pointer to another routine, and then executing it when appropriate. By generic design we mean not only vectored execution, but also a logical factoring of the job to be done so that the algorithm being written can be blind to the data on which it operates. Developers can then use these routines on great varieties of data types, formats and structures, even those never considered by the writers of the routines.

Overview

What does Forth need most, in order to become more widely useful and accepted in the computer industry? One of the most critical advances would be the widespread use of standard libraries of routines. We need the system software, documentation and shared conventions to support developers who can then take large modules of code — designed and programmed at various installations with different data formats and programming conventions — and re-use these modules in new contexts.

The modules should remain identical, usable with no changes at all, so that their users do not need to learn their internals, and do not risk introducing errors into software which may have been well tested through prior use at dozens or hundreds of installations.

One contribution to the development of standard libraries would be wider use of generic routines, when possible. For example, a formula evaluation might be defined first for single-precision arithmetic operations — add, subtract, multiply and divide — and then used unchanged for double-precision or complex numbers, or for other data entities. This flexibility requires (1) that the procedure make sense in its new domain and (2) that only the operators to be changed (the arithmetic, in the above example) know about their data; nothing else within the algorithm being programmed can know the length or format of the data items.

This article shows another example: sorting. Some sort algorithms can be defined in terms of only two operations (compare and exchange) which know about the data being sorted. Both operations take two arguments, indices or other pointers to the two items to be compared or exchanged. Comparison returns one result, a truth value; each operation may also return an error-test flag. The sort routine itself needs three arguments: pointers to the two routines, and the number of items to sort. It need know nothing about the format of the data.

Whoever uses the sort is responsible for defining the comparison and exchange operations for the particular data to be sorted. These definitions know the length and location of key fields, whether the sort is ascending or descending, etc. They must handle any resource management required, such as use of Forth buffers if the data is on disk, or memory management if variable-length records were being sorted.

Design Details

To simplify this article, we have illustrated it with an easy, exchange-sort algorithm, not an optimal method. Performance falls off sharply when many items must be sorted. (For a faster program, note Wil Baden's "Quicksort and Swords," *Forth Dimensions* VI/5. That program uses a generic design like the one presented here, although the user interface is different.)

Let's call the routine we are defining **SORT** and the operations it uses **COMPARE** and **EXCHANGE. SORT** will call these operations repeatedly, and must be able to tell them which items to compare or exchange; **COMPARE** and **EXCHANGE** must be able to find each item, given its position in the current sequence. We will use zero-origin indexing, requiring that **COMPARE** and **EXCHANGE** accept arguments zero through n–1, where n is the number of items to be sorted. **COMPARE** should return a true flag if the items must be exchanged, false otherwise. Therefore, it should return false for the equal case, to avoid an unnecessary exchange.

For simplicity, we wrote this example program to allow up to 32K items. It could easily be expanded to use unsigned or double-precision arguments.

Optimization

The key challenge here is that we know nothing about the items being sorted. Still, some optimization can be planned.

Since exchanges might be expensive (for very long records, for example), we should avoid doing them unnecessarily. In this example, instead of doing a bubble sort, we find the minimum (or maximum) item, and then exchange it, once, into its final place.

The data may be in memory or on disk. If on disk, in most cases each item will fit within a single Forth buffer, instead of spanning buffers. Then we need at least two buffers for reasonable performance — one holding the minimum (or maximum) found so far, the other for the item with which it will next be compared. Note that **SORT** cannot keep the minimum item in memory in order to optimize, as it has no idea of the size of the object being sorted, or how to move it; or the size, location or structure of its key; or whether the object is on disk in the first place.

Error Control

SORT and also the **COMPARE** and **EXCHANGE** defined by the user, could each return an error flag to the stack; non-zero could indicate error. For example, **COMPARE** and **EXCHANGE** might flag an error if a data item were incorrect or unreadable due to disk failure. **SORT** might abort and return an error flag in that case. For this tutorial example, we have omitted error flags.

Examples of Use

Note that the sort is in screens two and three. The rest of the code shows examples.

Screens six, seven and eight each have one example: sorting fifty binary numbers in RAM in ascending sequence; sorting fifty 64-character records on disk (major key: columns 1-3, ASCII, ascending; minor key: columns 11-15, ASCII, descending); and sorting fifty entire Forth screens (key: columns 1-64, ASCII, ascending). Incidentally, the timings are 2, 17 and 625 seconds, respectively, using the F83 version of Forth-83 on an IBM PC with floppies. These poor showings result from the inefficient sort algorithm and the time to move data on the disk.

Note that in Forth-83, the "tick" operation (the single quote) must be replaced with **[']** if used inside a colon definition.

When you write **COMPARE** or **EXCHANGE** for items on disk, be careful to use the buffers properly. An absolute address within a Forth block buffer becomes unreliable after any other I/O is done, because the same block may then be assigned to a different buffer. Do not store such an address for later use. Instead, either go through **BLOCK** again to re-access the data later, or move the data out of the buffer into other memory and use it from there.

Future Improvements

The best way to improve this routine would be to use a more efficient sort algorithm. For tutorial purposes, the one given here is adequate.

This example **SORT** is not re-entrant; it uses ordinary variables to store its arguments. We suggest that developers of transportable library modules use local variables, rather than elaborate stack manipulation, to get re-entrant code. Local variables have not yet been standardized in Forth; see the *Proceedings of the 1984 Forml Conference* for some excellent papers on the subject.

Incidentally, we could make **SORT** run a little faster by eliminating the mechanism of sending addresses which then require use of **EXECUTE**. Instead, **COMPARE** and **EXCHANGE** could be defined and used by **SORT** like any other words in the dictionary. But some generality would be lost — for example, the ability to sort different kinds of data structures with the same object code.

Examples two and three show that **EXCHANGE** could easily be parameterized and made available as a utility. **EXCHANGE** might even be put inside the sort, which could then have a tem-

```
Scr # 1         A:FD.BLK
 \ Generic sort routine, Forth-83                               04Mar85 JJ

 \    This program will sort any randomly-accessible data,
 \ either in RAM or on disk.  To use it, you must write two
 \ routines: one to compare two of your data items, and the
 \ other to exchange two of them.

 \    To simplify this tutorial, we have used an exchange
 \ sort, which is inefficient for large numbers of items.
 \ Quicksort could be substituted for a production version.

 2 LOAD  3 LOAD   \ The sort

 \ Three examples are in screens 6, 7, and 8


Scr # 2         A:FD.BLK
  \ Generic sort: setup                                         17Mar85 JJ
 VARIABLE A-COMPARE?    \ Address of Compare routine
 VARIABLE A-EXCHANGE    \ Address of Exchange routine
 VARIABLE N    \ Number of items to be sorted

 : DO-COMPARE?    \ N1 N2 -- ?    ;P Compare two "items"
    A-COMPARE? @ EXECUTE ;
 : DO-EXCHANGE    \ n1 n2 --    ;P Exchange two "items"
    OVER OVER <>  IF  A-EXCHANGE @ EXECUTE
    ELSE  DROP DROP  THEN ;  \ Don't exchange item with itself


Scr # 3         A:FD.BLK
  \ Generic sort                                                17Mar85 JJ
 : FIND-MIN    \ n1 -- n2    ;P Find min (max) from n1 on
    N @ OVER  DO  \ Look at all items from n1 on
      DUP I DO-COMPARE?  IF  DROP I  THEN  \ Replace if new min
      LOOP ;
 : NSORT    \ --    ;P Ordinary case of sort, 3 or more items
    N @ 1- 0  DO  I FIND-MIN  I DO-EXCHANGE  LOOP ;
 : SORT    \ acompare aexchange n --    ;P Save arguments, test
    N !  A-EXCHANGE !  A-COMPARE? !
    N @ 1 >  IF  NSORT  THEN ;  \ If less than 2, we're done
```

porary memory area, perhaps a few hundred bytes or so, for efficiently exchanging long data items piece by piece. **SORT** would have to be given the record length in that case.

Not only **EXCHANGE** but **COMPARE** also could be moved inside the sort. But then all of the information about the keys would have to be passed to **SORT** — not only the record length. In this extreme case, our routine would have become an ordinary sort package. It would have lost its versatility, because it would have to embody assumptions about the data, instead of letting its users manage their own data by programming.

Importance

The simple sort routine given here may not convey the practical importance of generic design, because this program could easily be rewritten every time. But the sort could be much more elaborate; for example, it could scan the data and select the best of several algorithms. Either the sort and/or the routines passed to it could be partly or entirely in code, with no problem of compatibility between code and high-level.

The speed penalty for transferring control to outside routines appears to be insignificant, even if an all-code generic program is compared with a special-purpose sort written entirely in code. The significant cost of using the generic design approach is that not all algorithms can be written in terms of **COMPARE** and **EXCHANGE**, or any other predefined set of operations. In many cases this cost will be worth paying.

Note that Forth gives us the flexibility to design modular program elements within the continuum between finished application packages and special-purpose programs written from scratch. Few higher-level languages encourage users to pass a subroutine to a module, which then executes that subroutine without knowing anything about its data.

Other Similar Aproaches

Many programming languages use systems of data abstraction or hiding to separate modules, reducing complexity and the chances of error by preventing side effects. In most of these systems, the subroutine knows about the data, but the calling program does not. Here, the roles are changed. The calling program knows about the data, and it passes a module which also knows about the data into a subroutine, which does not know about that data but executes the module at appropriate times. The module communicates with the subroutine by its normal input and output, and it communicates with the calling program by directly affecting its data, as it was designed to do. Other language constructs relevant to this approach include the "generic procedures" of ADA (which are templates resolved at compile time) and the "operators" of APL (which accept routines as arguments — for example, the inner-product operator accepts + and * to perform matrix multiplication).

Forth is more extensible than these languages, and it offers a key advantage of very low expense for experimentation. We can quickly put programming concepts to the test. Practical program modularization presents unsolved problems. Useful results, not fixed rules known in advance, serve as the guides in this effort.

```
Scr # 6          A:FD.BLK
\ Example 1: Sort 50 binary numbers in RAM            04Mar85 JJ
CREATE DATA 100 ALLOT
: X   \ n -- a   ;P Get address of nth element in DATA array
   2 *   DATA + ;
: COMPARE   \ n1 n2 -- ?   ;P Compare two items, given item #s
   SWAP X @  SWAP X @  > ;   \ Ascending, so exch if 1st is >
: EXCHANGE   \ n1 n2 --   ;P Exchange two items
   DUP X @  ROT ROT  \ Save a copy of one value, 3rd on stack
   OVER X @  SWAP X !  \ Move the other value into place
   X !  ;  \ Move the copy into place
: SORT-TEST1   \ --   ;P Sort the array
   ['] COMPARE  ['] EXCHANGE  50 SORT ;
\ Note: if test from keyboard, use  '  , not  [']



Scr # 7          A:FD.BLK
\ Example 2: Sort 64-character records                04Mar85 JJ
\ Note: uses F83 string compare,  COMP  a1 a2 n -- -1|0|+1
10 CONSTANT START-BLOCK  \ First block taken as 64-char records
: X   \ n -- a   ;P Get address of nth data element
   64 *  1024 /MOD  START-BLOCK +  BLOCK  + ;
: COMPARE   \ n1 n2 -- ?   ;P Compare two items, given item #s
   X  PAD 15 CMOVE  \ Get one key out of block buffer
   X  DUP PAD 3 COMP ( Major keys )   ?DUP 0= IF  \ Need minor
     DUP 10 +  PAD 10 +  5 COMP  NEGATE  THEN
   SWAP DROP ( Arg )   1 = ;  \ If +1, from either key, exch
: EXCHANGE   \ n1 n2 --   ;P Exchange two items, given item #s
   DUP X  PAD 64 CMOVE   OVER X  PAD 64 +  64 CMOVE
   X UPDATE  PAD 64 +  SWAP 64 CMOVE  \ Can't move buf to buf
   X UPDATE  PAD SWAP 64 CMOVE ;
: SORT-TEST2   \ --   ;P Sort 50 64-byte records on disk
   ['] COMPARE ['] EXCHANGE 50 SORT FLUSH ;

Scr # 8          A:FD.BLK
\ Example 3: Sort entire Forth screens                04Mar85 JJ
\ Note: uses F83 string compare,  COMP  a1 a2 n -- -1|0|+1
10 CONSTANT START-BLOCK  \ First of the blocks to be sorted
: X   \ n -- a   ;P Get address (in buffer) of nth block
   10 +  BLOCK ;
: COMPARE   \ n1 n2 -- ?   ;P Compare two blocks, first 64 char
   X PAD 64 CMOVE  \ Get one out of the buffer
   X PAD 64 COMP   1 = ;
: EXCHANGE   \ n1 n2 --   ;P Exchange two blocks
   DUP X  PAD 1024 CMOVE   OVER X  PAD 1024 +  1024 CMOVE
   X UPDATE  PAD 1024 +  SWAP 1024 CMOVE
   X UPDATE  PAD SWAP 1024 CMOVE ;  \ Note need 2K bytes at PAD
: SORT-TEST3   \ --   ;P Sort 50 screens
   ['] COMPARE ['] EXCHANGE 50 SORT FLUSH ;
```

A Forth Spreadsheet

Craig A. Lindley
Manitou Springs, Colorado

This article presents the implementation of a spreadsheet program written entirely in high-level Forth. It is based on the Laxen and Perry F83 model. People wishing to implement this program in other dialects of Forth will have to modify it accordingly.

The spreadsheet presented here does not claim to contain all the fancy features provided by the majority of spreadsheet programs in the commercial market. It was developed as an example program to illustrate structured programming techniques. It does, however, support the following features:

- 26 columns by 26 rows
- auto-calculation mode
- algebraic input of cell equations
- full-screen editing
- unlimited expansion
- data replication

(See table one for descriptions of supported commands.)

It is important to understand that the basic spreadsheet presented here could be expanded to have all of the features of the more exotic spreadsheet programs on the market. A very important result of structured program design is the ease of modification to and maintenance of the program. Once the structure of this program is understood, modification should be an almost trivial task. To help with the understanding of this program, the pseudo-code design from which it was coded is included herein. [*Due to the length of this article, the forty-five screens of source code were deferred to the following issue. —Ed.*]

Spreadsheet Commands

COMMAND	DESCRIPTION
C	Allows input of column names. Mode terminated by entering a <CR> to the program prompt.
A	Replicate cell data. This will copy data from the currently selected cell (< >) for the specified number of columns.
D	Input cell data. If auto-calculate mode in effect, spreadsheet will be recalculated automatically.
E	Input cell equation. Input terminated by <CR>.
F	Change number entry/display format. Normal or dollars and cents format.
G	Go to the specified row column. Selected cell is made current.
M	Change spreadsheet mode. Auto-calculate or normal.
N	New. Clears existing spreadsheet. All names equations and data are deleted.
O	Change calculation order. Either rows then columns or columns then rows.
P	Perform spreadsheet calculations. Forces recalculation.
Q	Quit spreadsheet.
R	Input row names. Mode terminated by entering a <CR> to the program prompt.
-->	Move current cell one position left.
<--	Move current cell one position right.
/\	Move current cell one positon upwards.
\/	Move current cell one position downwards.
control -->	Move to last column and display.
control <--	Move to first column and display.
Pg Up	Move four columns left.
Pg Dn	Move four columns right.
home	Go to first row and display.
end	Go to last row and display.

NOTES:

1. All alphabetic commands are processed by the Forth word command_in. All others are processed by control_in.

Table One

Program Operation

We will concentrate our attention at this time on the operation of the program. To compile the program under Laxen and Perry's F83 after you have entered it, simply type:

```
open spread.blk <cr>
1 load <cr>
```

```
A. Spreadsheet display before data entry


                                   Forth Spreadsheet
          Title
---------------------------A--------------B--------------C--------------D------
                        0|       0 |          0 |<         0>|          0 |
                        1|       0 |          0 |          0 |          0 |
                        2|       0 |          0 |          0 |          0 |
                        3|       0 |          0 |          0 |          0 |
                        4|       0 |          0 |          0 |          0 |
                        5|       0 |          0 |          0 |          0 |
                        6|       0 |          0 |          0 |          0 |
                        7|       0 |          0 |          0 |          0 |
                        8|       0 |          0 |          0 |          0 |
                        9|       0 |          0 |          0 |          0 |
                       10|       0 |          0 |          0 |          0 |
                       11|       0 |          0 |          0 |          0 |
                       12|       0 |          0 |          0 |          0 |
                       13|       0 |          0 |          0 |          0 |
                       14|       0 |          0 |          0 |          0 |
-------------------------------------------------------------------------------


B. Typical spreadsheet after data entry


                                   Forth Spreadsheet
          Title             jan           feb          mar           apr
---------------------------A--------------B--------------C--------------D------
income                  0|  $2750.00 |  $2750.00 |< $2750.00>|  $2750.00 |
                        1|     $0.00 |     $0.00 |     $0.00 |     $0.00 |
loan principle          2|    $63.45 |    $63.45 |    $63.45 |    $63.45 |
loan interest           3|   $757.83 |   $757.83 |   $757.83 |   $757.83 |
loan insurance          4|    $37.50 |    $37.50 |    $37.50 |    $37.50 |
                        5|     $0.00 |     $0.00 |     $0.00 |     $0.00 |
car payment             6|   $200.61 |   $200.61 |   $200.61 |   $200.61 |
car gas                 7|   $120.34 |   $200.21 |   $360.32 |   $105.63 |
car misc maint.         8|     $0.00 |    $20.00 |     $0.00 |    $45.60 |
                        9|     $0.00 |     $0.00 |     $0.00 |     $0.00 |
utilities              10|   $230.54 |   $230.54 |   $230.54 |   $230.54 |
savings account        11|   $500.00 |   $500.00 |   $500.00 |   $500.00 |
                       12|     $0.00 |     $0.00 |     $0.00 |     $0.00 |
cost of life           13|  $1910.27 |  $2010.14 |  $2150.25 |  $1941.16 |
money left             14|   $839.73 |   $739.86 |   $599.75 |   $808.84 |
-------------------------------------------------------------------------------

See text for details.
```

Figure One

which will start the process. At this time, the screen will clear and the message

Spreadsheet Compiling

will appear. The F83 system prompt "ok" will reappear when the compilation is completed. To execute the spreadsheet program, type:

spreadsheet <cr>

and you will see the display shown in figure one-a. Notice that at any one time, the display shows four columns and fifteen rows of the 26 x 26 spreadsheet. Every row/column intersection is referred to as a cell of the spreadsheet. Further, the cell surrounded by the greater-than/less-than symbols is called the "current cell." Data and/or equations can only be entered at the current cell.

Positioning of the current cell is controlled by the cursor arrow keys and the G (or Go-To) command. If the current cell position tries to leave the display window, the window will scroll to keep the current cell position on the display. See table one for a list of all commands used for display positioning.

As an example of how this spreadsheet is used, let's construct a simple home budget sheet. Figure one-b shows how this might look when we are finished. The first step in building a new spreadsheet is to give the various rows and columns names. The column names shown in the figure correspond to the months of a year. The various row names are shown on the left of the figure.

Column names are input to the spreadsheet program by selecting "C" from the command menu. This command will prompt for the column letter at which to begin the naming, and then for each of the desired names. For our example, enter (starting at column A) the three-letter abbreviations for the twelve months of the year, each followed by a <cr>. After inputting "dec" for December, hit <cr> twice to exit the column-name entry mode.

Row names are entered in exactly the same manner. The row-name entry mode is selected via the "R" command. If you wish to leave a blank line for a particular row, enter a space followed by a <cr>. Entering just a <cr> will terminate this mode of operation.

Because our budget spreadsheet will be used for monetary quantities, we must select the dollars/cents format for our display. This is accomplished by selecting the "F" — for Format — command and then selecting the dollars/cents mode. You will notice the display now shows "$0.00" for each entry, instead of just "0".

To place data into our spreadsheet, use the cursor positioning keys to place the current cell at row 0, column A, if it is not there already. Select the "D" command to enter data at this location. Enter "2750 <cr>" (the trailing decimal is implied). In this example, our income is assumed to be constant from month to month. Use the "A" command to enter this data again for the eleven subsequent columns of this spreadsheet. If you use the cursor positioning keys to move the display window around on the spreadsheet, you'll notice $2750.00 is entered as the first entry in each month.

Loan principle, interest, insurance, car payment, utilities and saving deposit are also the same amount for each month, so enter them in the same manner. Quantities that change from month to month, like car maintenance and gasoline, must be entered separately, using the "D" command described above.

The final two rows on the spreadsheet — money left and cost of life — are calculated items. By this, I mean they are dependent on other amounts already entered in the spreadsheet, and will require equations to be entered for these quantities. Using the cursor positioning keys, position the current cell at 13A in preparation for equation input. Now select the "E" — or Equation — command to input the following equation:

2 A + 3 A + 4 A + 6 A + 7 A + 10 A + 11 A <cr>

Note: The spaces between the characters are very important for

```
                 Psuedo-code for Forth Spreadsheet

PROCEDURE SPREADSHEET            (spreadsheet)
output initial screen display (dis_screen)
do forever
        get operator input          (IBM_key)
        if control then
                process control input (control_in)
        else
                process command input (command_in)
        endif
        display current status      (dis_status)
enddo


PROCEDURE PROCESS CONTROL INPUT (control_in)
do case of control instruction
        home key: do top row                    (top_row)
     up arrow: do up arrow                  (up_arrow)
        Pg Up: do left 4 columns            (left_4_cols)
   left arrow: do left arrow                (left_arrow)
  right arrow: do right arrow               (right_arrow)
          end: do bottom row                (bottom_row)
   down arrow: do down arrow                (down_arrow)
        Pg Dn: do right 4 columns           (right_4_cols)
 ^ left arrow: do first column              (first_col)
 ^ rght arrow: do last  column              (last_col)
else
          error condition (beep)
endcase
return


PROCEDURE COMMAND INPUT (command_in)
do case of operator command
        A: replicate cell data               (again_repl)
        C: input column names                (input_col_names)
        D: input cell data                   (input_cell_data)
        E: input cell equations              (input_equ)
        F: input number display format       (format)
        G: goto specified cell               (go_to)
        M: set calculate mode                (mode)
        N: clear spreadsheet                 (new)
        O: set calculation order             (order)
        P: perform calculations              (perform_calc)
        Q: quit spreadsheet                  (quit_calc)
        R: input row names                   (input_row_names)
else
           error condition (beep)
endcase
return

PROCEDURE GO TO (go_to)
prompt for row number

if within proper range then
        prompt for column letter
        if within proper range then
                make the specified row/col the current one
                set row/col displacement to zero
                display the data on display (dis_data)
        endif
endif
return
```

proper operation of the equations. If a mistake is made entering an equation, hit <cr> and then select "E" again and re-input the equation.

Next, use the down-arrow cursor positioning key to move the current cell down one position. Input the equation:

0 A - 13 A <cr>

This equation subtracts income from our expenses to give us the amount left over. This amount will always be displayed in cell 14A. Use this same technique for each of the twelve monthly columns.

After all data entry is completed, the spreadsheet can be calculated by executing the "P" — or Perform calculation — command. Before your eyes, you will see the totals for each month displayed. Scroll the spreadsheet to see each month's totals. To perform "what if" types of analysis, select the auto-calculate mode via the "M" — or Mode — command. This will force recalculation of the complete spreadsheet every time new data is entered. For example, decrease your February income (using the "D" command) and watch the result in cell 14B. Even this simplistic example program demonstrates the power of this program for real-world situations.

All commands supported by this spreadsheet program, as mentioned previously, are shown in table one. You might notice the absence of a command to print the spreadsheet on a printer. This feature could easily be added, or you can use the screen-print utility provided by many operating systems to make hard copy when necessary.

To save a spreadsheet for further use, type the following:

' spreadsheet is boot <cr>
save-system filename.com <cr>

This will create a stand-alone program called filename.com (or any other name you would like to give a .com file) that will execute immediately upon typing

filename <cr>

This spreadsheet program now has become a part of the F83 system and will execute (with all data and

```
PROCEDURE REPLICATE CELL DATA (again_repl)
get data of currently marked cell
prompt operator for number of columns to copy data into(#in)
if number of columns is greater than 0
        do for the specified number of columns
                move cell marker right one cell (right_arrow)
                copy data into cell
        enddo
        display data on screen (dis_data)
endif
return


PROCEDURE FORMAT (format)
output format prompt to operator
get response
if = 1 then
        set format flag true
else
        set format flag false
endif
return


PROCEDURE PERFORM CALCULATIONS (perform_calc)
calculate cells (calc_cells)
display the data on the display (dis_data)
return


PROCEDURE MODE (mode)
output mode command prompt
get response
if = 1 then
        set mode flag true
else
        set mode flag false
endif
return


PROCEDURE NEW (new)
ask again (y/n)
if answer is yes then
        clear cells array
        clear row name array
        clear col name array
        erase all equations from dictionary
        set row/col displacement to zero
        display the data on the display (dis_data)
endif
return


PROCEDURE QUIT (quit_calc)
ask again (y/n)
if answer is yes then
        abort program (abort)
endif
return
```

equations intact) immediately upon loading.

Modifications for Your Computer

If you have an IBM-compatible computer, this program will run without modification. Most other computers will need the key codes changed, however, to accommodate those returned by your system. Specifically, the spreadsheet words **IBM_key** (defined in screen seven), **control_in** (defined in screen forty-four) and, finally, **spreadsheet** (defined in screen forty-five), will need to be modified.

IBM_key is an IBM-specific word that allows access to all 256 of the key codes returned by the IBM keyboard driver. It maps the "extended key codes" produced by the PC into the range 128 - 256 decimal to allow easy access by the programmer. The **control_in** word case statement is based upon these key codes. In your system, first determine what key codes you wish to use to access the functions selected with **control_in** and then edit them into screen 44. Also, screen 45 will have to be changed to select either **control_in** or **command_in** in accordance with the range of key codes you have chosen. After the appropriate changes to the key codes are made, the program should compile and run without difficulty.

The coding of this spreadsheet program is a relatively straightforward process, given the finished design in pseudo-code. Two aspects of this implementation need to be discussed to make clear the operation of the program. These are (1) data structures utilitized and (2) algebraic equation usage.

Data Structures

Arrays are used for the data structures in this spreadsheet program. Two types — two dimensional and string arrays — are used to satisfy the data storage requirements of this program. A two-dimensional array called "cells" is used to hold all information about a particular cell of the spreadsheet. As defined in screen 6

```
PROCEDURE INPUT EQUATION (input_equ)
prompt for equation input
move definition preamble to terminal input buffer (tib) area
let operator input equation following preamble
move definition post-amble to tib
store total definition length in #tib to make forth
 think it all came from the operator
interpret equation definition into dictionary
 using algebra vocabulary
reselect forth vocabulary
return

PROCEDURE INPUT CELL DATA (input_cell_data)
prompt for data to be entered at currently marked cell
get input data (get#)
store into marked cell
get mode flag
if auto calculation mode selected then
        calculate all cells (calc_cells)
endif
display data on display  (dis_data)
return

PROCEDURE GET INPUT DATA (get#)
input a number from the operator
get format flag
if dollars and cents format selected then
        do case of decimal point position
                no decimal: multiply number input by 100
                 1 decimal: multiply number input by  10
                 2 decimal: multiply number input by   1
                 3 decimal: divide   number input by  10

        endcase
endif
return

PROCEDURE INPUT COLUMN NAMES (input_col_names)
prompt operator for starting column letter (A-Z)
make it the current column (one displayed in upper left)
do from the current column till final column
   output column identification letter
   input column name from operator into column name array
   if entry = CR (no name input)
           undo      (exit procedure)
   endif
   if 4 names have been input
           scroll display right to show them (dis_col_change)
   endif
   display column names (dis_col_names)
enddo
return
```

of the listing, each entry in the cells array (row,col) is six bytes in depth. The six-byte data sub-structure is organized as follows:

0 - 1 Equation CFA storage
2 - 5 Double Integer Value storage

Bytes 0 and 1 contain the code field address (CFA) of an equation, if one has been assigned to this cell. Zeros are stored in these locations if no equation exists. Bytes 2, 3, 4 and 5 contain storage for a double-length integer that is the current value of this particular cell. Specifying a particular row and column can, therefore, pinpoint in the cells array not only a cell's value, but also its defining equation.

Two string arrays — **col_names** and **row_names** — are defined for storage of the user-specified column and row names. As with all arrays used in this program, an index value on the parameter stack followed by the array name will result in the array element's address being returned to the top of the stack. For example:

```
PROCEDURE INPUT ROW NAMES (input_row_names)
prompt operator for starting row number
make that row the current row
do from specified row to maximum row
        display row prompt
        get row name from operator
        store name in row name array
        if only CR entered
                undo    (exit procedure)
        endif
        if 5 row names have been entered
                scroll screen vertically (dis_row_change)
        else
                display tow names
     endif
enddo
return


PROCEDURE START ALGEBRAIC DEFINITION (a[)
set operator stack to empty
select algebra vocabulary
return

PROCEDURE RIGHT PARENTHESIS ())
do while items on operator stack
        pop operator stack
        compile operator into forth dictionary (op>)
enddo
if left parenthesis found then
        backup operator stack pointer by 4 to remove it
else

        display "Missing (" error message
        abort program
endif
return

PROCEDURE LEFT PARENTHESIS (()
place CFA of )missing routine on top of operator stack
place a precedance of 1 on the   top of operator stack
push both onto the stack (>op)
return

PROCEDURE INFIX (infix)

HIGH LEVEL DEFINITION - compile
get CFA of double integer math routine
place precedence on top of parameter stack
store both into high level definition

HIGH LEVEL DEFINITION - runtime at equation compile time
get CFA and precedence from high level definition to parameter
stack if higher precedence than operator on top of operator
  stack then place CFA and precedence on top of operator stack
else
        compile operator into definition
endif
return

PROCEDURE END ALGEBRAIC DEFINITION (]a)
pop remaining items off operator stack and compile (op>)
select forth vocabulary
return
```

3 col_name

will return the address of column name four (remember, array elements are numbered from zero) to the top of the stack. Also:

3 4 cells 2+ 2@

will return the double integer value of the cell at the intersection of row 3 and column 4 to the top of the stack. The CFA of this cell's equation, if one exists, can be accessed by

3 4 cells @

If a value other than zero is returned, the cell has been assigned an equation. The equation can be executed, with the final result being placed in the same cell, as follows:

3 4 cells calculate

See the listing for the definition of **calculate**. The spreadsheet words **calc_r/c** and **calc_c/r** use this technique for stepping through the spreadsheet and calculating each cell's value.

Algebraic Equations

To make the spreadsheet easier to understand and use, it was decided during the design phase to make all equations input by the operator in algebraic — as opposed to reverse Polish (RPN) — form. Suppose the current cell on the display (the one surrounded with the < > characters) is 3A, and you want it to contain the sum of cells 0A, 1A and 2A. By selecting the input equation command "E" from the menu, you could enter:

0 A + 1 A + 2 A <cr>

From this time forward, the displayed value of cell 3A will reflect the sum of cells 0A, 1A and 2A after each time the spreadsheet is recalculated. The algebraic operators currently supported are +, -, *, / and mod, although other operators could be added easily by use of the technique shown in screens 30 and 31.

The words involved in algebraic equation processing are contained in screens 27 - 32 and 37 of the listing. Their operation is described somewhat in the program's design. The method utilized here was conceived by Michael Stolowitz (*Forth Dimensions* IV/6).

Basically, the program word **input_equ** builds an equation in the terminal input buffer (TIB) area in the form:

```
: FORMULA a[ ---------- ]a
[ cell_ptr
2+ ] literal 2! ;
last @ name> cell_ptr !
```

where the area denoted by hyphens is the algebraic equation input by the operator. When the operator enters a carriage return, the entire equation is compiled into the Forth dictionary with the name **FORMULA**. The symbol **a[** informs the compiler that an algebraic equation follows which will be terminated by **]a**. The next portion of the equation, up to and including the semicolon, stores the double-integer result left on the stack by the algebraic equation into the storage area of the cell corresponding to the equation just entered. The final portion of the equation returns the CFA of the equation just entered for storage into the CFA storage area for this cell. The end result of this process is that whenever the CFA is executed, the compiled equation will be executed, with the result being stored back into the corresponding cell. (You will note that each equation stored in the Forth dictionary is given the name **FORMULA**. This does not matter, as each is executed via its CFA and not by its name.) For additional information on parsing of algebraic equations, see the article mentioned above.

```
PROCEDURE RIGHT FOUR COLUMNS (right_4_cols)
do from 0 to 4
        right arrow (right_arrow)
enddo
return


PROCEDURE LEFT FOUR COLUMNS (left_4_cols)
do from 0 to 4
        left arrow (left_arrow)
enddo
return


PROCEDURE BOTTOM ROW (bottom_row)
current row = 11    (max row - 15)
scroll display vertically (dis_row_change)
return


PROCEDURE TOP ROW  (top_row)
current row = 0    (top row)
scroll display vertically (dis_row_change)
return


PROCEDURE LAST COLUMN (last_col)
current column = W  (max col-4)
scroll display horizontally (dis_col_change)
return


PROCEDURE FIRST COLUMN (first_col)
current column = 0  (first column)
scroll display horizontally (dis_col_change)
return


PROCEDURE DOWN ARROW (down_arrow)
get current marked cell position
if at bottom of display then
        if not at last row possible then
                increment current row number
                scroll display vertically (dis_row_change)
        endif
else
        erase cell marker (erase_cell_marker)
        increment row displacement from current row
endif
place cell marker on display (place_cell_marker)
return


PROCEDURE UP ARROW (up_arrow)
get current marked cell position
if cell is at top of display then
        if not at top of spreadsheet then
                decrement current row number
                scroll display vertically (dis_row_change)
        endif
else
        erase cell marker (erase_cell_marker)
        decrement row displacement from current row
endif
place cell marker (place_cell_marker)
return
```

```
PROCEDURE LEFT ARROW (left_arrow)
get current marked cell position
if at left end of display then
        if not at first column of spreadsheet then
              decrement current column number
              scroll display horizontally (dis_col_change)

        endif
else
        erase cell marker (erase_cell_marker)
        decrement column displacement from current column
endif
place cell marker (place_cell_marker)
return


PROCEDURE ORDER (order)
output operator prompt
get response
if = 1 then
        set order flag true
else
        set order flag false
endif
return


PROCEDURE CALCULATE ALL CELLS (calc_cells)
get order flag
if set
        calculate columns and then rows (calc_c/r)
else
        calculate rows and then columns (calc_r/c)
endif
return



PROCEDURE CALCULATE COLUMNS AND THEN ROWS (calc_c/r)
do from the first to the last row
        do from the first to the last column
                get cell formula address (CFA)
                calculate formula (calculate)
        enddo
enddo
return


PROCEDURE CALCULATE ROWS AND THEN COLUMNS (calc_r/c)
do from the first to the last column
        do from the first to the last row
                get cell formula address (CFA)
                calculate formula (calculate)
        enddo
enddo
return

PROCEDURE CALCULATE CELL FORMULA (calculate)
get data at cell formula address
if not equal to 0    (i.e. formula assigned for this cell)
        execute formula
endif

return
```

```
PROCEDURE DISPLAY SCREEN (dis_screen)
clear display
place cursor on CRT
display spreadsheet title
display boarder          (dis_boarder)
display menu             (dis_menu)
display column labels    (dis_col_labels)
display column names     (dis_col_names)
display row labels       (dis_row_labels)
display row names        (dis_row_names)
display data on display  (dis_data)
set row/col displacement to zero
display status           (dis_status)
return


PROCEDURE SCROLL DISPLAY HORIZONTALLY (dis_col_change)
display column names     (dis_col_names)
display column labels    (dis_col_labels)
display data on display  (dis_data)
return


PROCEDURE SCROLL DISPLAY VERTICALLY (dis_row_change)
display row names        (dis_row_names)
display row labels       (dis_row_labels)
display data on display  (dis_data)
return
PROCEDURE DISPLAY CURRENT STATUS (dis_status)
place cursor on CRT
display current row number
place cursor on CRT
display current column letter
place cursor on CRT
get mode flag
if set then
        display "AUTO"   i.e. auto calculate mode selected
else
        display "NORMAL"
endif
place cursor on CRT
get order flag
if set then
        display "C/R"    i.e. calculate columns then rows mode
else
        display "R/C"
endif
place cursor on CRT and display command prompt
place cell marker on display  i.e. place <   > around
                         currently selected cell
return
```

(Cont. on page 25)

October 23, 1985 — November 3, 1985

FORML

Forth Modification Laboratory
presents

EuroFORML Conference

Stettenfels Castle

Heilbronn, Federal Republic of Germany
Followed by

SYSTEMS Trade Fair, Munich

Computers and Communications 9th International
Trade Fair and International User's Congress
and
Selected sightseeing tours and entertainment in Germany

International technical conference October 25–27, 1985 Stettenfels Castle

Software Metrics — Programs and methods to measure program performance, complexity, structure, programmer productivity, development methods, models, tools, program verification aids, and procedures. Individual participation is encouraged and attendees are requested to submit a conference paper. Conference proceedings will be published.

SYSTEMS Trade Fair October 28 — November 1, 1985 Munich Fair Grounds

Computers and Communications — This is a major international event covering computers and communications. The trade fair is scheduled October 28 through November 1, 1985.

Guest and Tour Program — A complete program will be available for guests not attending the technical conference sessions. Sightseeing escorted tours are planned for the group.

Reservations, authors instructions, itinerary, special group rate — Write to **EuroFORML,** Forth Interest Group, Post Office Box 8231, San Jose, CA 95155 or telephone the FIG Hotline (408) 277-0668. East and West Coast departures are planned. **Advance reservations are required.**

CALL FOR PAPERS

FORTH INTEREST GROUP MAIL ORDER FORM

P.O. Box 8231 San Jose, CA 95155 (408) 277-0668

Please Print

Name ____________________

Company ____________________

Address ____________________

City ____________________

State/Prov. __________ ZIP __________

Country ____________________

Phone ____________________

Membership in the FORTH Interest Group
& Volume 7 of FORTH Dimensions
No sales tax, handling fee or discount
on membership. See col. 3 ☐

BACK VOLUMES

Volume 1 FORTH Dimensions	*$15/16/18	____
Volume 2 FORTH Dimensions	15/16/18	____
Volume 3 FORTH Dimensions	15/16/18	____
Volume 4 FORTH Dimensions	15/16/18	____
Volume 5 FORTH Dimensions	15/16/18	____
Volume 6 of FORTH Dimensions	$15/16/18	____

REFERENCE

FORTH 83 Standard	$15/16/18	____
FORTH 79 Standard	15/16/18	____
Bibliography of Forth References, 2nd Ed.	15/16/18	____

OFFICE USE ONLY

By ______ Date ______ Type ______
Shipped by ______ Date ______
UPS Wt. ______ Amt. ______
USPS Wt. ______ Amt. ______
BO ______ Wt. ______ Amt. ______

BOOKS ABOUT FORTH

All About FORTH	*$25/26/35	____
Beginning FORTH	17/18/21	____
Complete FORTH	16/17/20	____
FORTH Encyclopedia	25/26/35	____
FORTH Fundamentals, V. 1	16/17/20	____
FORTH Fundamentals, V. 2	13/14/16	____
FORTH Tools	19/21/23	____
Learning FORTH	17/18/21	____
Mastering FORTH	18/19/22	____
Starting FORTH (Soft Cover)	20/21/22	____
Thinking FORTH (Soft Cover)	16/17/20	____
Thinking FORTH (Hard Cover)	23/25/28	____
Threaded Interpretive Languages	23/25/28	____
Understanding FORTH	3.50/5/6	____

CONFERENCE PROCEEDINGS

FORML Proceedings 1980	$25/28/35	____
FORML Proceedings 1981 (2 V.)	40/43/45	____
FORML Proceedings 1982	25/28/35	____
FORML Proceedings 1983	25/28/35	____
FORML Proceedings 1984	25/28/35	____
Rochester Proceedings 1981	25/28/35	____
Rochester Proceedings 1982	25/28/35	____
Rochester Proceedings 1983	25/28/35	____
Rochester Proceedings 1984	25/28/35	____

JOURNAL OF FORTH APPLICATIONS AND RESEARCH

Journal of FORTH Research V. 1 #1	$15/16/18	____
Journal of FORTH Research V. 1 #2	15/16/18	____
Journal of FORTH Research V. 2 #1	15/16/18	____
Journal of FORTH Research V. 2 #2	15/16/18	____
Journal of FORTH Research V. 2 #3	15/16/18	____

REPRINTS

Byte Reprints	$5/6/7	____
Popular Computing 9/83	5/6/7	____
Dr. Dobb's 9/82	5/6/7	____
Dr. Dobb's 9/83	5/6/7	____
Dr. Dobb's 9/84	5/6/7	____

HISTORICAL DOCUMENTS

Kitt Peak Primer	$25/27/35	____
fig-FORTH Installation Manual	15/16/18	____

***PRICING**
Column 1 - US, Canada, Mexico
Column 2 - Foreign Surface Mail
Column 3 - Foreign Air Mail
Prices Are Subject To Change.

ASSEMBLY LANGUAGE SOURCE LISTINGS

1802	*$15/16/18	____
6502	15/16/18	____
6800	15/16/18	____
6809	15/16/18	____
8080	15/16/18	____
8086/88	15/16/18	____
9900	15/16/18	____
ALPHA MICRO	15/16/18	____
Apple II	15/16/18	____
ECLIPSE	15/16/18	____
IBM/PC	15/16/18	____
NOVA	15/16/18	____
PACE	15/16/18	____
PDP-11	15/16/18	____
VAX	15/16/18	____
Z80	15/16/18	____

MISCELLANEOUS

T-Shirt Size: ______	$10/11/12	____
Poster (BYTE Cover)	15/16/18	____
Handy Reference Card	FREE	☐

Subtotal	$ ____
10% Member Discount Member No. Required ________	____
Subtotal	**$** ____
CA Residents Add Sales Tax	____
Handling Fee	$2.00
Membership (*20/27/33)	____
TOTAL	**$** ____

☐ VISA ☐ Mastercard # ____________
Expiration Date ____________

$15 Minimum on VISA/Mastercard Orders.

Make check or money order payable in US funds drawn on a US Bank to: FIG.

PAYMENT MUST ACCOMPANY ALL ORDERS (Including Purchase Orders).
All Prices Include Shipping.

5/85

```
PROCEDURE DISPLAY COLUMNS NAMES (dis_col_names)
place cursor on CRT
do from current column four times
        if current column = max column
                undo  (exit procedure)
        endif
        place cursor on CRT
        get column name from column name array
        display at proper position
enddo
return


PROCEDURE DISPLAY COLUMN LABELS (dis_col_labels)
do from current column four times
        if current column = max column
                undo  (exit procedure)
        endif
        place cursor on CRT
        generate alphabetic label
        display at proper position
enddo
return


PROCEDURE DISPLAY ROW NAMES (dis_row_names)
do from current row 15 times
        if current row = max row
                undo  (exit procedure)
        endif
        place cursor on CRT
        get row name from row name array
        display at proper position
enddo
return


PROCEDURE DISPLAY ROW LABELS (dis_row_labels)
do from current row 15 times
        if current row = max row
                undo  (exit procedure)
        endif
        place cursor on CRT
        generate row number
        display at proper position
enddo
return


PROCEDURE DISPLAY DATA ON'DISPLAY (dis_data)
do for all 4 possible screen display columns
        if column displayed = final column number
                undo
        endif
        do for 15 possible screen display rows
```

```
                    if row displayed = final row number
                            undo
                    endif
                    position cursor on CRT
                    get cell content at cells[row,column]
                    format cell data (format#)
            enddo
enddo
return

PROCEDURE FORMAT CELL DATA (format#)
get format flag
if set
        format as dollars/cents (fd.r)
else
        format as number (d.r)
endif
return

PROCEDURE ERASE CELL MARKER (erase_cell_marker)
calculate cell display location (cal_cell_disp_loc)
unmark cell (unmark_cell)
return
PROCEDURE PLACE CELL MARKER (place_cell_marker)
calculate cell display location (cal_cell_disp_loc)
mark cell (mark_cell)
return


PROCEDURE ASK AGAIN (Y/N)
place cursor on CRT
display "Are you sure" message
get response
convert to upper case character
if yes then
        set result true
else
        set result false
endif
return
```

NOTES:

a. The words shown in parentheses are the Forth words that were coded from the pseudo code design. Refer to listing one for the actual code generated from this design.

The Mirth Dimension...

```
: HEN-SCRATCH   ( n --- ? )
    RANDOM   0 DO
              PICK  PICK  PICK  PICK  PICK
                  DROP  DROP  EMIT  EMIT
    LOOP  ;


: FUNCTIONAL-SPEC   ( pages  ---  ?? )
    1000  *
    0  DO
         I  HEN-SCRATCH
    LOOP  ;
```

--Scott Heiner & Charles Knowlton

Macro Generation

Don Taylor
Sydney, Australia

In a past issue of *Forth Dimensions* (V/5), Jeffrey Soreff presented a method of writing macros in Forth. The idea was to put **COMPILE** before each non-immediate word, **[COMPILE]** before each immediate word, and make the whole thing **IMMEDIATE**. This certainly does the job but, of course, it leads to definitions in which every second word is **COMPILE** or **[COMPILE]**. Inspired by Soreff's article, I set myself the task of writing a defining word that would create a macro from any segment of legal Forth code. A solution is presented in figure one.

Typing:

MACRO: <name>

creates a dictionary entry for <name> and copies all the text following <name> up to the next semi-colon into the parameter field. The dictionary entry is completed by inserting ¦ (offset by blanks) after the text.

When <name> is encountered within a colon definition, it redirects the input stream to the text within its parameter field. Then, **INTERPRET** compiles the words that it finds there as though they were part of the colon definition. The input stream is restored to its original state by the word ¦ that occurs at the end of each macro.

It is possible to nest the macros created by this approach, and it is not necessary to have defined any of the words within the macro at the time of its creation. Of course, these words do need to be defined before the macro is used.

This solution to the problem has an obvious drawback. Namely, it consumes a large amount of dictionary space. On the other hand, it does allow a great deal of freedom and, since the macros are not needed after they have been used, space could be saved by loading them as **TRANSIENT** definitions (see the note by Phillip Wasson, *Forth Dimensions* III/6) and removing them after compilation of the words that use them.

If compilation crashes within a macro, **TIB** will be left pointing somewhere inside the dictionary. To restore normal input, use **TIB!** from figure one and **FORGET** the corrupted macro definition.

The macros **DO'** and **LOOP'** given in figure two correspond in function to the macros with the same names provided by Jeffrey Soreff.

```
: ASCII    BL WORD 1+ C@ STATE @ IF [COMPILE] LITERAL THEN ; IMMEDIATE

: MACRO:   CREATE   ASCII ; WORD C@ BL C, ALLOT ASCII ¦ C, BL C,

                    IMMEDIATE

           DOES>    R> BLK @ >R >IN @ >R TIB @ >R >R TIB !

                    0 BLK ! 0 >IN ! ;

: ¦        R> R> TIB ! R> >IN ! R> BLK ! >R ; IMMEDIATE

TIB @  CONSTANT  TIB@

: TIB!     TIB@ TIB ! ;
```

Figure One

```
MACRO: DO'      2DUP - 0> IF DO ;

MACRO: LOOP'    LOOP ELSE 2DROP THEN ;


: EXAMPLE       CR DO' I . CR LOOP' ;          ( Macro example )

: EXAMPLE       CR 2DUP - 0>                   ( Equivalent code )

                IF DO I . CR LOOP ELSE 2DROP THEN ;
```

Figure Two

(cont.)

```
3 LIST
Screen # 3
 CR ." The MACRO generator "
: MACRO:  CREATE   ASCII ; WORD C@ BL C, ALLOT ASCII ¦ C, BL C,
                   IMMEDIATE
          DOES>    R> BLK @ >R >IN @ >R TIB @ >R >R TIB !
                   0 BLK ! 0 >IN ! ;
: ¦       R> R> TIB ! R> >IN ! R> BLK ! >R ; IMMEDIATE

  TIB @   CONSTANT TIB@
: TIB!    TIB @ TIB ! ;

( Example.   cf. Forth Dimensions  V/5 )
MACRO: DO'      2DUP - 0> IF DO ;
MACRO: LOOP'    LOOP ELSE 2DROP THEN ;

: EXAMPLE       CR DO' I . CR LOOP' ;

 Ok
3 LOAD
The MACRO generator  Ok
EXPAND EXAMPLE
   CR
   2DUP
   -
   0>
   OBRANCH  18
   <DO>
   I
   .
   CR
   <LOOP>  -8
   BRANCH  4
   2DROP
   ;S  Ok
3 4 EXAMPLE
 Ok
4 3 EXAMPLE
3
 Ok
```

Figure Three

Keywords — Where Used

Nicholas L. Pappas
Oakland, California

```
                                        (x = stack bottom)
CR   CR   ."  LOOKING FOR  "            cfa  n  addr  x
DUP  0  4  D.R
SPACE
DUP  2+  NFA  ID.
```

Figure One

```
CR                     cfa  n  addr  x
ROT  ROT                 n  addr  cfa  x
OVER  +             addr+n  addr  cfa  x
SWAP                addr  addr+n  cfa  x
```

Figure Two

```
                                    addr  addr+n  cfa  x
DO                                                cfa  x
    DUP  I  @                             ni  cfa  cfa  x
    =                                          f  cfa  x
    IF  I  0  4  D.R  SPACE                       cfa  x
         I  FINDID.                               cfa  x
    THEN
LOOP                                              cfa  x
```

Figure Three

```
BEGIN                              addr  x
         DUP  XLIT  =          f1  addr  x
         SWAP  1  -         addr-1  f1  x
```

Figure Four

```
DUP  1+  @            ni  addr-1  f1  x
DOCOL  =              f2  addr-1  f1  x
```

Figure Five

```
DUP                           f2  f2  addr-1  f1  x
IF@                               f2  addr-1  f1  x
       OVER  1+             addr  f2  addr-1  f1  x
       0  4  D.R
       SPACE
THEN                              f2  addr-1  f1  x
```

Figure Six

We have created a number of tools to facilitate our work: **FINDNO** is one such tool. **FINDNO** tells which keywords use a given keyword. For example, when one wants to load Forth above 8000h in memory, you quickly discover the need to replace < with **U**< so that addresses, which are unsigned numbers, are compared correctly. Or, when base changes are annoying, you may want to ask, "Which keywords change base, and where are those keywords used?" Suddenly, you need to know which keywords use <, **BASE**, **HEX** and **DECIMAL**.

The basis for **FINDNO** is this: when keyword A uses keyword B, A's code body includes B's code field address (cfa). So we need to search memory for the two-byte cfa number starting at some address for some number of bytes. Consistent with Forth memory-reference keywords, the prefaces

addr, number-of-bytes, cfa

give **FINDNO** the data it needs to do its task. **FINDNO** starts searching at *addr* for *number-of-bytes*, looking for *cfa* (keyword B) in order to reveal keywords A using B.

Proceeding in a simple way, we read each byte pair (*addr @*) while incrementing *addr* by one, not two. This means we search through memory from Forth's start address to the dp value. Since we read through name bytes and link field address bytes, as well as the code bytes, we take the risk of getting false reports. Incrementing *addr* by one avoids the complicating questions, "Where are the code cells, and does this Forth use byte cells (naughty, naughty) as well as word cells?"

How does **FINDNO** work? With the cfa on top of the stack, the initial code fragment shown in Figure One leaves the stack values alone as it prints a friendly message telling us what is about to happen (e.g., "Looking for 1624 Compile."

Then we start a new line and manipulate the stack values to calculate a loop index and limit as shown in Figure Two. We have a known number of bytes to search, so a do loop that increments by unity is what we use. The loop index I is an address because the loop limits are addresses. Note the consequent simplicity in Figure Three.

```
   ROT                        f1   f2   addr-1   x
    OR                             f3   addr-1   x
UNTIL                                   addr-1   x
```

Figure Seven

```
3   +                    addr+2   x   (forward to pfa)
NFA  ID.                          x
CR   ;
```

Figure Eight

```
              NLF FORTH EDITOR
SCR: 220      INSERT OFF
   ^-------^-------^-------^------^
CR ." scr utility 820512"

' : CFA @ CONSTANT DOCOL
' LIT LFA CONSTANT XLIT
: FINDID.    ( addr ---    )
   BEGIN
     DUP XLIT =
     SWAP 1 -
     DUP 1+ @
     DOCOL =  DUP
     IF OVER 1+ 0 4 D.R SPACE
     THEN
     ROT
     OR
   UNTIL
   3 +
   NFA ID.
   CR ;

: FINDNO ( addr n1 n2 ---     )
   CR CR ." LOOKING FOR "
   DUP 0 4 D.R
   SPACE DUP 2+ NFA ID.
   CR ROT ROT OVER + SWAP
   DO DUP I @ =
      IF I 0 4 D.R SPACE
         I FINDID.
      THEN
   LOOP
   DROP ; ;S
COPYRIGHT (C) 1983
by Nicholas L. Pappas, PhD
```

I @ is *addr* @ that leaves *ni*, which is compared to *cfa* so that flag *f* is non-zero if equal and zero if not equal. The if-then statement is skipped on false flags, **LOOP** increments the index by one and branches back to **DUP** for a look at the next byte pair. On true flags, the if-then statement executes to print the address holding a number equal to the cfa of B, leaves the address on the stack and executes **FINDID.**. **FINDID.** assumes the number is indeed a cfa being used by a code body as it proceeds to print the cfa and <name> of the using keyword (keyword A). More later on **FINDID.**.

Our useful friend *cfa* is still on the stack, so we end with **DROP** ;.

The basis for **FINDID.** is that docol — run-time code for : — is stored in keyword A's cfa. (Only colon definitions have cfa's in their code bodies, so this is real.) If the number *ni* is not really a cfa, then it is in an lfa or part of a <name>. **FINDID.** still moves down memory through the next code body, looking for docol, and performs its tasks — producing a false report. (More later on false reports.) In the unlikely, yet possible, event there are no docols down memory, **FINDID.** does nothing and exits gracefully when **LIT**'s lfa is reached. If **LIT** is not your first keyword, redefine **XLIT** accordingly.

Here is how **FINDID.** works. Not knowing *a priori* where docol is, we use a begin-until loop for our search. We do last things first in order to avoid some stack manipulation and to be easier to read ("think-about ... until"). First, test an exit possibility by checking for end-of-search and backing up one byte to the code in Figure Four.

In case our cfa is also docol we just left it, so the code in Figure Five follows. If *f2* is true, we execute the if-then statement, printing the cfa of user A with Figure Six. Checking for an exit, we get both flags on top and do a logical-or operation, as in Figure Seven, to exit if *f3* is true (non-zero) or to loop if it is false. When we exit, note that *addr-1-1* is the lfa (*lfa* = *addr-2* if *addr* is the cfa of the keyword). We want to print the <name> of our user A via **ID.** so we need its nfa (see Figure Eight).

```
6E0 20 DUMP

06E0 26 00 83 42 4C CB D7 06 17 06 16 00 82 42 D3 E2
06F0 06 17 06 36 00 87 44 49 53 50 43 4F CC EC 06 17
  OK
```

```
' COMPILE CFA . 1606   OK
0100 1F00 1606 FINDNO

LOOKING FOR 1606 COMPILE
 5E2  5DE ;
 6E9  668 :
167A 1676 ;CODE
16DD 16D3 LITERAL
1B60 1B5A AGAIN
1BB1 1B7F DO
1BB3 1BAD ELSE
1BCE 1BCC IF
1BE9 1BE3 LOOP
1BFF 1BF9 +LOOP
1C15 1C0F UNTIL
1DED 1DDF ."
  OK
   OK
' [COMPILE] CFA . 1624   OK
0100 1F00 1624 FINDNO

LOOKING FOR 1624 [COMPILE]
  OK
PCRT
```

```
0100 1F00 ' EMIT CFA FINDNO

LOOKING FOR  944 EMIT
 C02  BA6 EXPECT
 C34  C30 SPACE
 CAC  C98 TYPE
1945 191B INDEX
19D0 199C TRIAD
  OK
   OK
0100 1F00 ' KEY CFA FINDNO

LOOKING FOR  954 KEY
 BB0  BA6 EXPECT
1EB9 1EA7 KX
  OK
   OK
0100 1F00 ' ?TERMINAL CFA FINDNO

LOOKING FOR  96A ?TERMINAL
17AA 1762 VLIST
1935 191B INDEX
19B6 199C TRIAD
  OK
PCRT
```

Figure Nine

In the examples, **COMPILE**'s "where-used list" includes colon. This is a false report, because the value 1606h (**COMPILE**'s cfa) happened to be *within* a user variable. So **FINDID.** backed up past douser (no docol in a user variable) and kept going until it found a docol — this happened to be in colon. The clue is the large difference, for a keyword, between the two printed addresses 668 and 6E9 (see the memory dump in Figure Nine).

Note that the simple test **DOCOL =** in **FINDNO** can be replaced by an or test for docol, dovar, docon, douser or dodoes; we let it go, in the interests of simplicity. Also note that the immediate word **[COMPILE]** does not show up as expected. And, perhaps a review of where **EMIT.**, **KEY** and **TERMINAL** are used is of interest. Finally, please note that a screen editor can be written which has a reformattable display complete with window roll-up and roll-down.

Simple Control of Search Order:

Not ONLY But ALSO

Bill Stoddart
Middlesbrough, England

The story so far:

"The evolution of Forth continues, particularly in the area of vocabularies. The latest step is a recognition of the importance of controlling the search order." *Bill Ragsdale, 1982 FORML Conference*

"The ONLY Concept for Vocabularies" was submitted by Bill as an experimental proposal in the Forth-83 Standard.[1,2] It departs somewhat from the standard and from other systems (including fig-FORTH and polyFORTH), in that executing a vocabulary name places that vocabulary at the start of the search order list, rather than actually specifying a search order. This paper argues that such a departure is not necessary. On the contrary, the standard forms a good basis for a set of simple and powerful words that give the Forth user complete control of the search order.

Vocabulary handling in my 83-Standard system is extended with four simple words: **SEARCHES**, **ALSO**, **END-SEARCH** and **SEAL**. These are all one-line definitions. They give complete control over the specification of search order.

Consider the creation of a new vocabulary with the phrase:

```
VOCABULARY APPLICATION
```

When **APPLICATION** is subsequently executed, it specifies a search order of **APPLICATION** followed by **FORTH**.

Suppose we want **APPLICATION** to specify a search order of **APPLICATION** followed by **MENU** followed by **EDITOR** followed by **FORTH**. This is achieved by the phrase:

```
APPLICATION  SEARCHES  MENU ALSO
EDITOR ALSO FORTH END-SEARCH
```

The specified search order becomes operational when **APPLICATION** is subsequently executed.

As this sequence of words is interpreted, the system **CONTEXT** is changing at a furious rate. Indeed, the fact that the vocabularies are actually executing their run-time behavior makes the definition of the search order setup words so simple. The still point in this storm is the **FORTH** vocabulary. New vocabularies are defined within the **FORTH** vocabulary, and since **SEARCHES** and **ALSO** both set **CONTEXT** to **FORTH**, the following vocabulary name is always "in context" (i.e., within the search order specified by **CONTEXT**).

Finally, the word **SEAL** is used to limit the search order specified by a vocabulary to that vocabulary's definitions, as in:

```
MENU SEAL
```

Subsequent execution of **MENU** sets up a search order containing a single vocabulary, which is **MENU** itself.

A problem arises when a sealed vocabulary is to be included in a search order setup sequence. Just consider the above setup sequence with **MENU** as the sealed vocabulary. After **MENU** executes, **ALSO** will be "out of context." There are ways around this, of which the most obvious is to compile the setup sequence before executing it, as in:

```
:  SETUP
APPLICATION SEARCHES MENU
ALSO EDITOR
ALSO FORTH
END-SEARCH  ;      SETUP
```

I leave the reader to think of a slightly less flexible alternative which requires no compilation!

DEFINITIONS is present with its usual usage, and **FORGET** can work across multiple vocabularies. ROMmable code is easily supported, though the definitions given here operate from RAM.

Example Application

Some of the most demanding control of search order occurs during metacompilation, but that is another story. The following example is a simple but realistic one involving the Forth assembler.

One of the best uses of vocabulary switching in Forth occurs in **CODE** definitions. **CODE** switches the context vocabulary to **ASSEMBLER**, and the words **IF**, **ELSE**, **THEN**, etc. take on meanings appropriate to code assembly. The default search order specified by a standard definition of **ASSEMBLER** would be **ASSEMBLER** then **FORTH**, but it can be useful to modify this. Suppose we have an application that interfaces to a network with portions of assembler code that need direct access to constants and data structures in a **NETWORK** vocabulary. Part of the application might be organized like this:

```
VOCABULARY  NETWORK
ASSEMBLER  SEARCHES  NETWORK
    ALSO  FORTH END-SEARCH
NETWORK DEFINITIONS
    CREATE BUFFER 256 ALLOT ( space
    for buffer ) HEX
E000 CONSTANT PORT-ADDRESS etc. . . .
```

The search order specified by **ASSEMBLER** (and therefore implicitly specified by **CODE**) has been set to **ASSEMBLER**, then **NETWORK**, then **FORTH**. We could now enter **CODE** definitions which contain references to words in the **NETWORK** vocabulary; for example:

```
CODE  SEND  (  ---  send packet )
PORT  #
DI  MOV BUFFER  #  SI  MOV etc. . . .
```

(This example is from an 8086 assembler. An I/O port is being moved into the DI register and a buffer address into the SI register.)

When the **NETWORK DEFINITIONS** are all loaded, we can restore

NEW BOOKS - "FORTH GUIDE" by Haydon. An exploration of the intricacies of MVP-FORTH $20.00
"FORTH Applications" by Weber $13.00

```
SCR 70
( Vocabularies )

: VOCABULARY
   CREATE   VOC @   HERE VOC !   DUP , ( compile VLINK )
   2+ C@   1+   C, ( next voc )   1 C, ( FORTH )   0 C,
   0 ,  DOES>  2+ CONTEXT !  ;

: SEARCHES ( -- addr1 addr2 )   CONTEXT @ 1+   FORTH  ;

: ALSO ( addr -- addr+1 )  CONTEXT @ C@   OVER C!  1+  FORTH  ;

: END-SEARCH ( addr1 addr2 -- )  ALSO 0 SWAP C!   FORTH  ;

: SEAL  0   CONTEXT @   1+ C!   FORTH ;

: DEFINITIONS   CONTEXT @   CURRENT !  ;
```

ASSEMBLER to its original meaning with:

ASSEMBLER SEARCHES FORTH END-SEARCH

Implementation

The parameter field of each vocabulary is a data structure which contains information to specify the search order. The words **SEARCHES**, **ALSO**, **END-SEARCH** and **SEAL** operate on these data structures. Another element in a vocabulary's parameter field is the **VLINK** field, which contains a pointer to the previously defined vocabulary. This information is used when creating the parameter field of a new vocabulary, and when **FORGET** operates across multiple vocabularies. The user variable **VOC** contains the address of the **VLINK** field of the most recently created vocabulary.

The following details and the source screens are particular to my own system, but the underlying ideas, as well as the glossary entries, are quite general.

Each vocabulary is identified by a number between one and sixteen. (A sixteen-thread hashing algorithm is used to organize the dictionary.[4]) A vocabulary's parameter field contains a list of up to four bytes which specify the search order. The value zero is used as a terminator. The number one identifies the **FORTH** vocabulary, and two the **ASSEMBLER** vocabulary. The seven-byte parameter field of **ASSEMBLER** looks like this:

VLINK 2 1 0 0 0

The two-byte field **VLINK** contains the address of the corresponding field in the dictionary entry for **FORTH**. A user variable **VOC** contains the address of the link in the most recently created vocabulary. This information is used to assign a number to the next vocabulary created.

If **FORTH** and **ASSEMBLER** are the only vocabularies in the system, and we now define:

VOCABULARY APPLICATION

its parameter field will contain:

VLINK 3 1 0 0 0

VOCABULARY uses **VOC** to locate the last vocabulary created, which was **ASSEMBLER**, and from this works out the new vocabulary's **VLINK** and number, which is three. **VOC** is updated to point to the **VLINK** field in the new vocabulary.

When **APPLICATION** becomes the **CONTEXT** vocabulary, **CONTEXT** holds the address of the third byte in the parameter field of **APPLICATION**. **FIND** scans this and the following bytes, and will search in turn vocabularies three and one.

Now we can walk through a typical search order setup:

ASSEMBLER Leaves **CONTEXT** pointing to the 2 in the parameter field of **ASSEMBLER**.

SEARCHES (--- addr) Leaves the adress of the following byte in the parameter field of **ASSEMBLER**.

APPLICATION Now **CONTEXT** points to 3 in the parameter field of **APPLICATION**.

ALSO (addr --- addr + 1) Copies the 3 from the **APPLICATION** vocabulary's parameter field into addr in the parameter field of **ASSEMBLER**.

FORTH Points **CONTEXT** to the **FORTH** vocabulary.

END-SEARCH Copies 1 (identifying **FORTH**) from the **FORTH** vocabulary's parameter field to the parameter field of **ASSEMBLER**, then writes 0 to the following address to mark the end of the search order list.

The parameter field of **ASSEMBLER** now contains:

LINK 2 3 1 0 0

APPLICATION becomes the **CURRENT** vocabulary in the usual way, by executing **APPLICATION DEFINITIONS**. **CURRENT** then holds the address of the third byte in the parameter field of **APPLICATION**. The contents of this location are used by **CREATE** to decide in which vocabulary new dictionary entries should be placed.

Suppose we want a search order containing more than four vocabularies? This is no problem. The additional bytes of the parameter field may be allotted when the vocabulary is created. Thus, if we wanted **APPLICATION** to eventually specify a search order of seven vocabularies, this would be set up with:

VOCABULARY APPLICATION 3 ALLOT

On the other hand, where memory is in short supply, we can recover unused bytes in a similar way.

Final Word

Four simple words have been added to an 83-Standard system to provide powerful facilities for the control of search order. These definitions will be easily adapted to systems in which the parameter field of a vocabulary entry contains information which directly specifies a search order. They provide facilities which are not available when search order is specified by the order in which vocabularies are created (as in the FIG model) and they provide a more readable source than systems such as polyFORTH which require the user to specify a search order in numeric format.

Glossary

ALSO (sys1 --- sys2) Set the **CONTEXT** vocabulary as the next vocabulary in the search order list identified by sys1. Leave sys2, which identifies the position of the following element in this list, for subsequent use by **ALSO** or **END-SEARCH**.

END-SEARCH (sys ---) Set the **CONTEXT** vocabulary as the next and final vocabulary in the search order list identified by sys.

SEAL Set the search order specified by the present **CONTEXT** vocabulary to contain only the present **CONTEXT** vocabulary, and make **FORTH** the new **CONTEXT** vocabulary.

SEARCHES (--- sys) Leave the system-dependent information sys which identifies the position of the first element in the **CONTEXT** vocabulary's search order list. Make **FORTH** the new **CONTEXT** vocabulary.

References

1. W.F. Ragsdale. "The ONLY Concept for Vocabularies." *1982 FORML Proceedings*.

2. W.F. Ragsdale. "Search Order Specification and Control." Experimental Proposal, *Forth-83 Standard*.

3. Evan Rosen. "Vocabulary Tutorial," Part two. *Forth Dimensions*, V/4.

4. M. McNeil. "Hashed Dictionary Searches." *1981 FORML Proceedings*, Vol. One.

Another Forth-83 LEAVE

John Hayes
Laurel, Maryland

I would like to propose yet another solution to the Forth-83 **LEAVE** problem. The ideal implementation of **LEAVE** should compile a **(LEAVE)** code primitive followed by a pointer to the first word after **LOOP** (or **+LOOP**), as in figure one. Since multiple **LEAVE**s are allowed per **LOOP** level, **LOOP** must somehow resolve all these forward references. Also, in nested **DO LOOP**s, **LEAVE** must exit only the innermost loop surrounding it. These requirements, combined with the fact that **LEAVE** will usually occur inside **IF THEN** control structures, suggest that the compile-time actions of **DO**, **LEAVE** and **LOOP** need to be quite complicated. However, the situation is not as bad as it seems.

My implementation is a modification of one used by Bill Stoddart (*Forth Dimensions* V/4). His solution avoids the problem of resolving multiple forward branches by having each of the **LEAVE**s point back to **DO**, where there is a pointer to the end of the **LOOP**. This is less efficient than the ideal implementation pictured in figure one. It turns out that coding the ideal solution is not difficult. I have written a general-purpose word **>>RESOLVE** that resolves multiple forward branches. I will explain how **>>RESOLVE** works in the context of the **LEAVE** problem. Then, to demonstrate the word's generality, I will show its application in a set of case structure compiling words.

In my implementation, a linked list of unresolved forward references is maintained. A **VARIABLE** named **CLUE** points to the most recent entry added to the chain. Each time the **IMMEDIATE** word **LEAVE** is executed, a code primitive **(LEAVE)** is compiled followed by a pointer back to the previous **LEAVE** link. If there are no previous **LEAVE**s, a null pointer is compiled. Then **CLUE** is updated to point to the new head of the list. It is **LOOP**'s job to convert this list into a set of pointers to the first word

```
( CODE FOR RESOLVING FORWARD AND BACKWARD BRANCHES )

: <MARK                              ( --- ADDR )    ( USED AS DESTINATION )
                                     ( OF BACKWARD BRANCH.                 )
 HERE ;

: <RESOLVE                           ( ADDR --- )    ( RESOLVE BACKWARD )
                                     ( BRANCH.                          )
 , ;

: >MARK                              ( --- ADDR )    ( SOURCE OF FORWARD )
                                     ( BRANCH.                           )
 HERE 2 ALLOT ;

: >RESOLVE                           ( ADDR --- )    ( RESOLVE FORWARD )
                                     ( BRANCH.                         )
 HERE SWAP ! ;

: >>RESOLVE                          ( OLDLINK --- )    ( RESOLVE A CHAIN )
                                     ( OF FORWARD BRANCHES.               )
 BEGIN
    DUP WHILE
    DUP @ HERE ROT !
 REPEAT DROP ;

( THE CODE WORDS [DO], [LOOP], AND [+LOOP] IMPLEMENT FORTH-83 DO..LOOPS. )
( [LEAVE] IS A FORTH-83 LEAVE. CLUE IS USED TO IMPLEMENT LEAVE.          )

 VARIABLE CLUE                       ( --- ADDR )    ( CLUE POINTS TO )
                                     ( LAST WORD IN LEAVE CHAIN.      )

: DO                                 ( --- CLUE HERE )
 COMPILE (DO) CLUE @ 0 CLUE ! <MARK ; IMMEDIATE

: LOOP                               ( CLUE HERE --- )
 COMPILE (LOOP) <RESOLVE
 CLUE @ >>RESOLVE
 CLUE ! ; IMMEDIATE

: +LOOP                              ( CLUE HERE --- )
 COMPILE (+LOOP) <RESOLVE
 CLUE @ >>RESOLVE
 CLUE ! ; IMMEDIATE

: LEAVE                              ( --- )
 COMPILE (LEAVE) HERE CLUE @ , CLUE ! ; IMMEDIATE
```

(**Listing One**)

```
( CASE SELECT COMPILING WORDS.  THE SYNTAX OF THE STRUCTURE IS:   )
(        : NUMCHECK                                                )
(          SEL                                                     )
(             << 0  ==> ZEROSTUFF MORESTUFF >>                     )
(             << 1  ==> ONESTUFF MORESTUFF  >>                     )
(             << 10 ==> TENSTUFF MORESTUFF  >>                     )
(             << OTHERWISE ==> OTHERSTUFF   >>                     )
(          ENDSEL ;                                                )

: SEL
 0 ; IMMEDIATE

: <<                                 ( OLDLINK --- OLDLINK )
 COMPILE DUP ; IMMEDIATE

: =>                                 ( --- IFADDR )
 COMPILE ?BRANCH >MARK
 COMPILE DROP ; IMMEDIATE

: ==>                                ( --- IFADDR )
 COMPILE =
 COMPILE ?BRANCH >MARK
 COMPILE DROP ; IMMEDIATE

: >>                                 ( OLDLINK IFADDR --- NEWLINK )
 COMPILE BRANCH SWAP ,
 >RESOLVE
 HERE 2- ; IMMEDIATE

: OTHERWISE                          ( --- )    ( [OPTIONALLY] CREATE )
                                     ( AN OTHERWISE CASE.             )
 COMPILE DUP ; IMMEDIATE

: ENDSEL                             ( OLDLINK --- )

 COMPILE DROP >>RESOLVE ; IMMEDIATE
```

(**Listing Two**)

after **LOOP**. This is where >>**RESOLVE** comes in. >>**RESOLVE**'s argument is a pointer to the start of a linked list. >>**RESOLVE** threads down the list, changing each pointer to **HERE** instead of the next link. Figure two-a shows a **DO LOOP** with two **LEAVE**s inside before **LOOP** is executed. Figure two-b shows the completed **DO LEAVE LOOP** structure.

The address of the **LEAVE** list has to be kept in a **VARIABLE** instead of on the stack. Since **LEAVE** can occur inside other control structures, a list address kept on the stack could be covered by an arbitrary number of words, making it impossible for **LEAVE** to find the address. But keeping the address in the **VARIABLE CLUE** introduces another problem. Each loop in a nested **DO LOOP** structure needs a separate **LEAVE** list. Therefore, at times there can be more than one unresolved **LEAVE** list. The solution is to have **DO** stack the old value of **CLUE** and store a new null pointer in **CLUE**. **LOOP**, after >>**RESOLVE**ing the current **LEAVE** list, will restore **CLUE** to its old value. This idea is due to Bill Stoddart.

Another instance where it is necessary to resolve multiple forward branches is in the case structure. The syntax of the structure is shown at the top of listing two. Each >> should compile a branch to the word following **ENDSEL**. The method of implementation is similar to the **LEAVE** list. Each time >> executes, it compiles a **BRANCH** primitive followed by a link to the previous >>. **ENDSEL** converts this linked list into pointers to **HERE** using >>**RESOLVE**.

Note that my Forth system used sixteen-bit absolute branches. If your system uses eight-bit relative branches, >>**RESOLVE** will be harder to code, but not impossible. Happy Forthing!

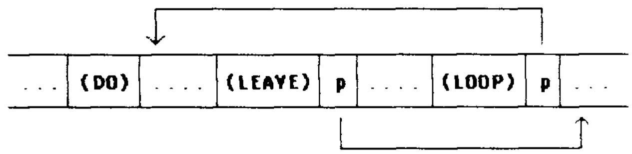


Figure One
Ideal DO...LEAVE...LOOP

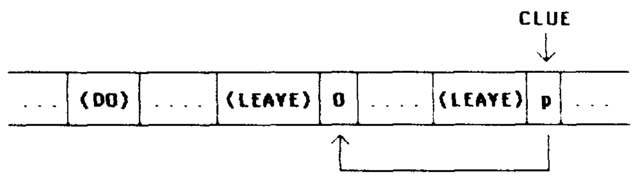


Figure Two a
DO...LEAVE...LEAVE...LOOP before
Loop is executed

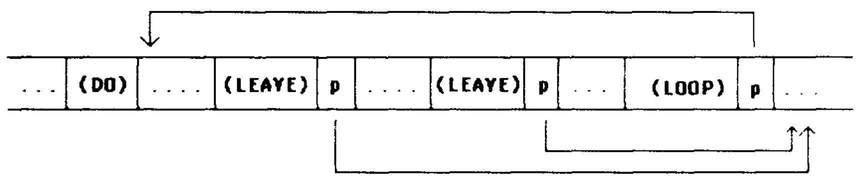


Figure Two b
DO...LEAVE...LEAVE...LOOP after
LOOP is executed

YACS* Part Two

*Yet Another Case Statement

Henry Laxen
Berkeley, California

Last time, we traced the history of the **CASE** statement in Forth and took a look at three different implementations of "indexed" **CASE** statements, namely **CASE** statements that were basically arrays of executable procedures. At run time, the index on the parameter stack was used to compute an index into this array, and the corresponding element of the array was executed. While this approach is often exactly what is required and is very efficient at run time, I pointed out that sometimes a more flexible **CASE** structure would be handy. I left you with a challenge, namely to come up with a **CASE** statement that adds the minimum number of new words to Forth and allows arbitrary Forth expressions to be used both as matching clauses and consequent clauses. My solution to this problem is presented in figure one, with examples of use in figure two. Let's take a look and see if we can figure out how it works.

First, let's look at the word **RUN** which, as the name implies, runs something. All it does is push the address that is on the parameter stack onto the return stack. This seems a bit suspicious, since we all remember from our early Forth training that we *never* push anything onto the return stack without later removing it in the same word; otherwise, disaster may result. Well, as in life, every rule was made to be broken. In this case, we use **RUN** to run a high-level code fragment. What happens is that the address we provide is pushed onto the return stack. Next, the **UNNEST** word compiled by ; executes, and pops the return stack into the **IP**. The net result is that interpretation proceeds at the address we provided on the parameter stack. When the **UNNEST** word at the end of the high-level code fragment is encountered, it will return to the word following the **RUN** in the high-level definition containing it. **RUN** would be a useful word to have in all Forth systems, since its virtue is that — unlike **EXECUTE** — it does not require a code field.

Now let's examine the word **CASE**. It works in conjunction with **END-CASE** as follows: **CASE** will compile high-level Forth phrases while the number on the top of the parameter stack is non-zero. Normally, the number on the parameter stack is the address of the beginning of the current code phrase, which should get resolved; however, when the word **END-CASE** executes, we notice that the first thing it does is a **DROP FALSE**, which will throw away the address and replace it with a zero. This will terminate the compilation loop. Notice also that **END-CASE** is an **IMMEDIATE** word, and hence executes even while compiling. The compile-time portion of **CASE** generates a linked list of code phrases. A picture illustrating this is in figure three, and represents the structure built in memory by the code in figure two. For those of you unfamiliar with the Forth-83 words **>MARK** and **>RESOLVE**, their definitions are as follows:

```
: >MARK HERE 0 , ;
: >RESOLVE HERE SWAP ! ;
```

Their function is to leave a pointer to a cell on the parameter stack and initialize the cell to zero, and to then resolve the contents of the cell whose address is on the stack to the current dictionary location. They are used extensively in the definitions of **IF ELSE THEN** and the looping words. They are also exactly what is called for here, to create a linked list in memory. The **!CSP** word is required for the compile-time error checking that is usually implemented inside ;.

Now then, let's analyze what is going on. At the beginning of the loop, we lay down a link address and call the Forth compiler with **]**. The Forth compiler compiles the following words in the input stream until it encounters a ;. The ; compiles an **UNNEST** for us and exits from the compiler. At this point, the address left by **>MARK** should still be on the stack; if it is, execution continues through the **WHILE**. The **>RESOLVE** word resolves the link left by the previous **>MARK** and branches back to the **BEGIN** to repeat the process. Thus, we are creating a linked list of code phrases, until the address that was placed on the stack by **>MARK** is replaced by a zero. This is done by **END-CASE**.

The run-time portion of **CASE** simply uses the information compiled by **CASE** to evaluate the first, third, fifth, etc. phrases and to compare them to the top of the parameter stack. If the value returned by the phrase equals the value on the stack, then the next phrase — an even-numbered one — is executed. If the values are not equal, the even phrase is skipped and the next odd phrase is executed. Notice that it is the user's responsibility to make sure that the phrases come in pairs, since **CASE** does no compile-time or run-time error checking. If we march all the way through the linked list and never find a phrase that generates a matching value, we will eventually encounter the zero link that was compiled last. This will cause us to exit the **BEGIN WHILE REPEAT** loop and **2DROP** throws away the initial value that was passed to us, and the zero that was fetched to terminate the list.

One interesting feature of this **CASE** statement is that in order to implement an **OTHERWISE** clause, which will always be executed if none of the previous clauses matched, we simply **DUP** the top of the stack. This will guarantee that the two values are equal, and the corresponding consequent clause will be executed.

```
\ GRAND CASE
: RUN   (S addr -- )   >R   ;
: CASE
  CREATE   (S -- )
  BEGIN   >MARK   !CSP   ]   DUP WHILE   >RESOLVE   REPEAT
  DOES>    (S n -- )
  BEGIN   DUP @ WHILE
     2DUP 2+ RUN  =  IF  NIP @ 2+ RUN  EXIT   THEN
     @ @ ( no match, link to next condition )
  REPEAT   2DROP   ;
: END-CASE  (S n -- 0 )
  DROP FALSE   [COMPILE] [   ;   IMMEDIATE
: OTHERWISE   (S n -- n n )   DUP   ;
```

Figure One

```
\ EXAMPLE OF A GRAND CASE
CASE REMARK
  2 ;  ." The only even prime" ;
  6 ;  ." The first perfect number" ;
  OTHERWISE ;  ." Nothing remarkable about it"  ;
END-CASES
  1 REMARK [CR] Nothing remarkable about it OK
  2 REMARK [CR] The only even prime OK
  6 REMARK [CR] The first perfect number OK
```

Figure Two

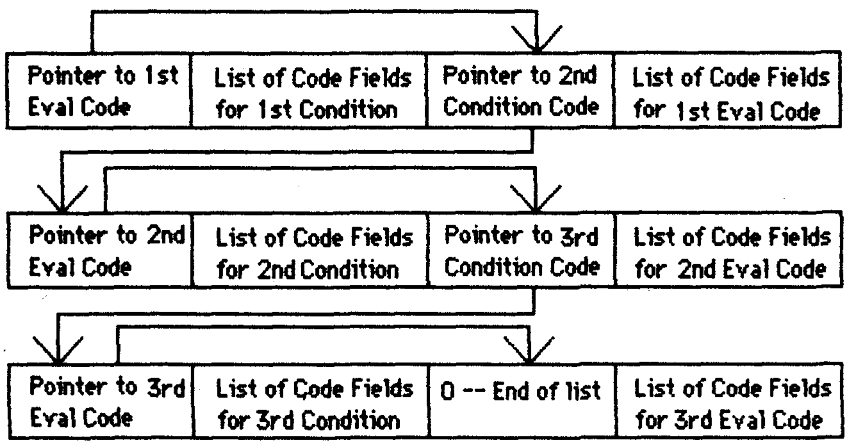


Figure Three

```
\ PROPOSED GRANDER CASE
: EQUAL   ['] = ;
: RANGE   ['] BETWEEN ;
HEX RANGE CASE CLASSIFY
    0 1F ;   ." Control Character" ;
   20 2F ;   ." Punctuation" ;
   30 39 ;   ." Digit "  ;
   3A 40 ;   ." Punctuation" ;
   41 5A ;   ." Upper Case Letters"   ;
   5B 60 ;   ." Punctuation"   ;
   61 7A ;   ." Lower Case Letters"   ;
   7B 7E ;   ." Punctuation"   ;
   7F 7F ;   ." Control Character"  ;
END-CASES
```

Figure Four

John D. Hall
Oakland, California

We want to welcome five new chapters:

Huntsville FIG Chapter,
Huntsville, Alabama

Central Iowa FIG Chapter,
Ames, Iowa

Fairfield FIG Chapter,
Fairfield, Iowa

North Orem FIG Chapter,
Orem, Utah

Lake Superior FIG Chapter,
Superior, Wisconsin

Central Connecticut FIG Chapter

Feb 6: On Wednesday, we met at the Meriden Public Library. Upon the suggestion of John Moran, work was begun on a test suite for fig-FORTH. As discussion continued on the subject, we realized we were taking on a nontrivial project. The purpose of this project is to give individuals who have versions of Forth a means of validating their instruction set. We are calling on the entire Forth community to help us! Although we intend to produce a program to validate the entire set of fig-FORTH words, we are aware that some versions of fig-FORTH, both commercial and public-domain versions, contain bugs. We would like to trap as many of these as we can. If any users out there can identify the bugs their versions contain, we would appreciate as much information as possible about these peculiarities so we can be sure these most common bugs are identified by the test suite. If you write us about an existing bug, please try to include: 1) the source of your Forth, 2) the date of release or version number, 3) the word(s) that don't work, 4) under what conditions this bug can be simulated, and 5) if known, the cause or a cure. Also, any references to prior work on this subject, or any other type of help at all, would be appreciated. Upon its completion, the test suite will be released — with much criticism, I'm sure — to the Forth community.

This is a very ambitious group project, and any Forth users in Connecticut who can help with suggestions or coding would be very welcome at our meetings! Please contact Charles Krajewski, 205 Blue Rd., Middletown, CT 06457, (203) 344-9996.

—*Charles Krajewski*

Atlanta FIG Chapter

Mar 19: Our meeting proved to be in our familiar mold — unstructured and with much exciting debate on various topics. Nathan Vaughn continued his explanations of ideas for an intelligent interest-matching system which will one day relieve him of much routine work. Anyone with knowledge of a method for counting word usage and managing a huge vocabulary, with elimination of infrequently used words, should contact Nathan. David Penz described his need for low-cost, PC-based productivity tools in a multitasking environment. Chuck Albert wants to apply Forth to the math used to predict the effect of complicated modulation on a carrier. Anyone with experience in using Forth on Bessel functions? To gain an overall impression of what the Forth community in Atlanta is doing, here are some of the topics I jotted down that came up in our conversations: 1) controller reading codes off of moulds, 2) epidemiology, 3) ultrasonics, 4) robotics, 5) color graphics, 6) fuzzy logic, and 7) bit-slice processors.

—*Ron Skelton*

Detroit FIG Chapter

Jan 22: Burce Bordt gave an interesting presentation of his interrupt-driven system. The system is operating on his homebrew 6809-based system. Except for two dependent machine-code words, the entire software system was written in high-level Forth. The system is written so that by changing a particular vector, execution of any word could be invoked by depressing a switch, triggered by a system timer, etc.

Feb 26: Randy White presented a short graphics "windowing" demo from the Val-Forth package on an Atari Computer. A continuing discussion of a graphics standard in Forth followed. A discussion also followed of the Bulletin Board System we have been trying to establish. The system would be used for message exchange, program exchange and announcements. Due to financial limitations at this time, it was decided to use an existing bulletin board or Compuserve for this purpose.

—*Tom Chrapkiewicz*

Hamburg FIG Chapter

Feb: The Hamburg chapter meets on the fourth Saturday of the month, and usually about twenty people show up. There are chapters forming in Berlin, Wuppertal, Kiel, Bremen, Paderborn and Karlsruhe. We are organizing "euroFORML 85," a multi-faceted conference on October 25-27 in a castle in southern Germany. Please plan to attend. See a call for papers elsewhere in this or the previous issue.

Orange County FIG Chapter

Jan 2: Wil Baden presented a calendar which easily calculates any day of the year. Roland Koluvek presented some work he had done over the holidays which, on a PC, allows you to leave Forth resident and return to DOS, then an ALT-Shift from DOS returns you to Forth. This is something like Sidekick. Allen Hansen had added some features to Leo Brodie's Quick Text Formatter.

Feb 6: Wil Baden presented a map of the United States done Forth style. Ken Clark presented a paper called "A Set of Formal Rules for Phrasing." These rules are regular and it is possible to pass raw code through a formatter and have it "phrased." Wil presented **:DOES>** which is his solution to the need in Forth for "self-defining words."

—Roland Koluvek

All the chapter hand-outs mentioned in these chapter reports that are sent to John Hall, are reproduced and redistributed to the chapters on a monthly basis. Check with your chapter for copies.

Silicon Valley FIG Chapter

Feb 23: We had about sixty people show up at the new meeting place in San Carlos. FORML used the library in the morning, and the afternoon FIG meeting used the gymnasium at the ABC School. The acoustics in the gym were bad, so we will try to use the library until we overflow. For the morning FORML session, Kim Harris suggested we organize some small working groups doing favorite projects that can be developed and presented as team efforts. We will select them next month. John James and Mike Ham discussed FIG's plan to distribute Forth material on the Delphi or Compuserve nets. Many FIG members already subscribe, and there are already Forth activities on these nets. FIG may be able to make these nets the focus of the exchange of Forth code and information, with the chapters as nodes to the members. A quick poll was conducted to see if members would discuss their projects and activities at work. Much work in Forth gets done on projects where Forth is not the main purpose of the project and is not visible. We would like to focus attention on these projects. Thirty people agreed, and each will be given time at the next meeting.

—John Hall

FIG Chapters

• ALABAMA

Huntsville FIG Chapter
Call Tom Konantz
205/881-6483

• ALASKA

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

• ARIZONA

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

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

• ARKANSAS

Central Arkansas Chapter
Twice Monthly: 2nd Sat., 2 p.m. &
4th Wed., 7 p.m.
Call Gary Smith
501/227-7817

• CALIFORNIA

Los Angeles Chapter
Monthly, 4th Sat., 10 a.m.
Hawthorne Public Library
12700 S. Grevillea Ave.
Call Phillip Wasson
213/649-1428

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

Orange County Chapter
Monthly, 4th Wed., 7 p.m.
Fullerton Savings
Talbert & Brookhurst

Fountain Valley
Monthly, 1st Wed., 7 p.m.
Mercury Savings
Beach Blvd. & Eddington
Huntington Beach
Call Noshir Jesung
714/842-3032

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

Sacramento Chapter
Monthly, 4th Wed., 7 p.m.
1798-59th St., Rm. A
Call Tom Ghormley
916/444-7775

Bay Area Chapter
Monthly, 4th Sat.
FORML: 10 a.m.
General: 1 p.m.
ABC Christian School Aud.
Dartmouth & San Carlos Ave.
San Carlos
Call: FIG Hotline — 415/962-8653

Stockton Chapter
Call Doug Dillon
209/931-2448

• COLORADO

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

• CONNECTICUT

Central Connecticut Chapter
Call Charles Krajewski
203/344-9996

• FLORIDA

Orlando Chapter
Every two weeks, Wed., 8 p.m.
Call Herman B. Gibson
305/855-4790

Miami
Monthly, Thurs., p.m.
Coconut Grove area
Call John Forsberg
305/252-0108

Tampa Bay Chapter
Monthly, 1st Wed., p.m.
Call Terry McNay
813/725-1245

• GEORGIA

Atlanta Chapter
Call Ron Skelton
404/393-8764

• ILLINOIS

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

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

Rockwell Chicago Chapter
Call Gerard Kusiolek
312/885-8092

• INDIANA

Central Indiana Chapter
Monthly, 3rd Sat., 10 a.m.
Call John Oglesby
317/353-3929

Fort Wayne Chapter
Monthly, 2nd Wed., 7 p.m.
Indiana/Purdue Univ. Campus
Rm. B71, Neff Hall
Call Blair MacDermid
219/749-2042

• IOWA

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

Central Iowa FIG Chapter
Call Rodrick A. Eldridge
515/294-5659

Fairfield FIG Chapter
Monthly, 4th day, 8:15 p.m.
Call Gurdy Leete
515/472-7077

• KANSAS

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

• LOUISIANA

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

• MASSACHUSETTS

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

• MICHIGAN

Detroit Chapter
Monthly, 4th Wed.
Call Tom Chrapkiewicz
313/562-8506

• MINNESOTA

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

• MISSOURI

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

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

• NEVADA

Southern Nevada Chapter
Call Gerald Hasty
702/452-3368

• NEW HAMPSHIRE

New Hampshire Chapter
Monthly, 1st Mon., 6 p.m.
Armtec Industries
Shepard Dr., Grenier Field
Manchester
Call M. Peschke
603/774-7762

• NEW MEXICO

Albuquerque Chapter
Monthly, 1st Thurs., 7:30 p.m.
Physics & Astronomy Bldg.
Univ. of New Mexico
Call Rick Granfield
505/296-8651

• NEW YORK

FIG, New York
Monthly, 2nd Wed., 8 p.m.
Queens College
Call Ron Martinez
212/517-9429

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

Syracuse Chapter
Monthly, 3rd Wed., 7 p.m.
Call Henry J. Fay
315/446-4600

• OHIO

Athens Chapter
Call Isreal Urieli
614/594-3731

Cleveland Chapter
Call Gary Bergstrom
216/247-2492

Cincinatti Chapter
Call Douglas Bennett
513/831-0142

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

• OKLAHOMA

Central Oklahoma Chapter
Monthly, 3rd Wed., 7:30 p.m.
Health Tech. Bldg., OSU Tech.
Call Larry Somers
2410 N.W. 49th
Oklahoma City, OK 73112

• OREGON

Greater Oregon Chapter
Monthly, 2nd Sat., 1 p.m.
Tektronix Industrial Park
Bldg. 50, Beaverton
Call Tom Almy
503/692-2811

• PENNSYLVANIA

Philadelphia Chapter
Monthly, 4th Sat., 10 a.m.
Drexel University, Stratton Hall
Call Melonie Hoag
215/895-2628

• TENNESSEE

East Tennessee Chapter
Monthly, 2nd Tue., 7:30 p.m.
Sci. Appl. Int'l. Corp., 8th Fl.
800 Oak Ridge Turnpike, Oak Ridge
Call Richard Secrist
615/693-7380

• TEXAS

Austin Chapter
Contact Matt Lawrence
P.O. Box 180409
Austin, TX 78718

Dallas/Ft. Worth
Metroplex Chapter
Monthly, 4th Thurs., 7 p.m.
Call Chuck Durrett
214/245-1064

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

• UTAH

North Orem FIG Chapter
Contact Ron Tanner
748 N. 1340 W.
Orem, UT 84057

• VERMONT

Vermont Chapter
Monthly, 3rd Mon., 7:30 p.m.
Vergennes Union High School
Rm. 210, Monkton Rd.
Vergennes, VT
Call Don VanSyckel
802/388-6698

• VIRGINIA

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

Potomac Chapter
Monthly, 2nd Tues., 7 p.m.
Lee Center
Lee Highway at Lexington St.
Arlington, VA
Call Joel Shprentz
703/860-9260

Richmond Forth Group
Monthly, 2nd Wed., 7 p.m.
154 Business School
Univ. of Richmond
Call Donald A. Full
804/739-3623

• WISCONSIN

Lake Superior FIG Chapter
Call Allen Anway
715/394-8360

FOREIGN

• AUSTRALIA

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

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

• BELGIUM

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

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

• CANADA

Nova Scotia Chapter
Contact Howard Harawitz
227 Ridge Valley Rd.
Halifax, Nova Scotia B3P2E5
902/477-3665

Southern Ontario Chapter
Quarterly, 1st Sat., 2 p.m.
General Sciences Bldg.
Rm. 312
McMaster University
Contact Dr. N. Solntseff
Unit for Computer Science
McMaster University
Hamilton, Ontario L8S4K1
416/525-9140 ext. 3443

Toronto FIG Chapter
Contact John Clark Smith
P.O. Box 230, Station H
Toronto, ON M4C5J2

• COLOMBIA

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

• ENGLAND

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

• FRANCE

French Language Chapter
Contact Jean-Daniel Dodin
77 Rue du Cagire
31100 Toulouse
(16-61)44-03

• GERMANY

Hamburg FIG Chapter
Monthly, 4th Sat., 1500h
Contact Horst-Gunter Lynsche
Common Interface Alpha
Schanzenstrasse 27
2000 Hamburg 6

• IRELAND

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

• ITALY

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

• REPUBLIC OF CHINA

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

• SWITZERLAND

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

SPECIAL GROUPS

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

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

FIGGRAPH
Call Howard Pearlmutter
408/425-8700

FORTH INTEREST GROUP PRESENTS

Forth National Convention

September 20 - 21, 1985

Complete conference program, educational seminars, and commercial exhibits.

Hyatt Rickeys in Palo Alto, California USA

---

euroFORML Conference

October 23, 1985 - November 3, 1985

International Technical conference at Stettenfels Castle
SYSTEMS Trade Fair in Munich
Guest and Tour Program in Germany

Complete group travel arrangements from USA to Germany and return. Air travel on Lufthansa Air Lines.

Forth Modification Laboratory

November 29, 1985 - December 1, 1985

A technical conference for advanced Forth practitioners.

Asilomar Conference Center

Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California USA

Complete information available from the Forth Interest Group.

FORTH INTEREST GROUP
P. O. Box 8231
San Jose, CA 95155

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

Address Correction Requested