FORTH Dimensions

Volume 6, Number 2

July/August 1984
$2.50

Procedural Arguments

Rehabilitation Applications

Simple Data Transfer Protocol

Forth Control Structures

Sixth Annual

Forth National Convention

November 16–17, 1984

Hyatt Palo Alto

4290 El Camino Real, Palo Alto, CA 94306 USA

Learn about Forth and make your life easier. The convention will show you how!

- **Exhibits**
- **Speakers**
- **Tutorials**
- **Vendor Meetings**
- **Panel Discussions**
- **Equipment Demonstrations**
- ***Discussion Groups***
- **Worldwide FIG Meeting**
- **Banquets**
- **Awards**

Forth is for everyone. The Forth computer language is used in video games, operating systems, real-time control, word processing, spread sheet programs, business packages, DBMS, robotics, engineering and scientific calculations and more.

Coverage of Forth applications, Forth-based instruments, Forth-based operating systems, and more.

Speak at the convention. Those wishing to participate and be speakers and/or panelists are urged to contact the program coordinator immediately. (Telephone the FIG hotline 415/962-8653.)

PROGRAM

FRIDAY, November 14

EXHIBITS ***Noon — 6 pm***

11:30 am	**Registration**
1 pm	**Forth Systems**
2 pm	**Data Base Developments**
3 pm	**Forth-Based Products**
4 pm	**Forth-Based Products**
5 pm	**32 Bit Systems**
6 pm	**Exhibits Close**

SATURDAY, November 17

EXHIBITS 9 am — 5 pm

10 am	**Forth Resources**
11 am	**Education**
Noon	**Lunch**
1 pm	**Forth Chips and Computers**
2 pm	**Business Applications**
3 pm	**Forth Chapters**
4 pm	**Forth-83 Standard, FORML Preview**
5 pm	**Exhibits Close**

BANQUET

7 pm Saturday — Reservation and payment required — $30.00

Convention preregistration is $10.00; or $15.00 at the door. Special convention room rates are available at the Hyatt Palo Alto. Telephone direct to Hyatt reservations by calling (800) 228-9000 and request the special Forth Interest Group Convention rates for November 16th and 17th.

The Forth Convention is sponsored by the Forth Interest Group (FIG). The Forth Interest Group is a non-profit organization of over 4800 members and 50 chapters worldwide, devoted to the dissemination of Forth-related information. FIG membership of $15.00/year ($27.00 overseas) includes a one-year subscription to **FORTH Dimensions**, the bimonthly publication of the group.

☐ Yes! I will attend the Forth Convention.

☐ Number of pre-registered admissions ____ × $10.00 each $ ____________

☐ Number of Banquet Tickets ____ × $30.00 each ____________

☐ Yes! I want to join FIG and receive **FORTH Dimensions** ($15.00 US, $27.00 foreign) ____________

TOTAL CHECK TO FIG $ ____________

☐ I want to exhibit; please send exhibitor information.

Name ________________________________

Address ________________________________

Company ________________________________

City ____________________ State ________ Zip ____________

Phone (   ) ____________________

Return to: **Forth Interest Group**, P.O. Box 1105, San Carlos, CA 94070 • **415/962-8653**

FORTH Dimensions
Published by the
Forth Interest Group
Volume VI, Number 2
July/August 1984

Editor
Marlin Ouverson

Production
Jane A. McKean, Et Al.

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

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

Symbol Table

Simple; introductory tutorials and simple applications of Forth.

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

Advanced; requiring study and a thorough understanding of Forth.

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

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

fig — Code and examples conform to fig-FORTH.

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

FORTH Dimensions

FEATURES

DEPARTMENTS

SUPER FORTH 64®

By Elliot B. Schneider

TOTAL CONTROL OVER YOUR COMMODORE-64™
USING ONLY *WORDS*

MAKING PROGRAMMING FAST, FUN AND EASY!

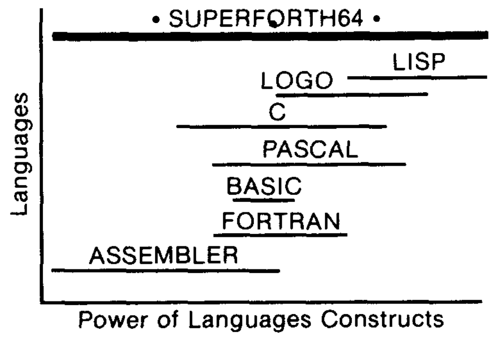
MORE THAN JUST A LANGUAGE...
A complete, fully-integrated program development system.

Home Use, Fast Games, Graphics, Data Acquisition, Business, Music
Real Time Process Control, Communications, Robotics, Scientific, Artificial Intelligence

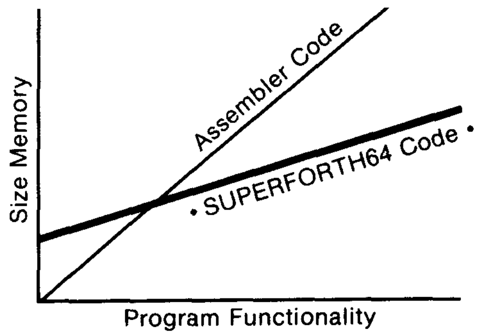
A Powerful Superset of MVPFORTH/FORTH 79 + Ext. for the beginner or professional

- 20 to 600 x faster than Basic
- 1/4 x the programming time
- Easy full control of all sound, hi res. graphics, color, sprite, plotting line & circle
- Controllable SPLIT-SCREEN Display
- Includes interactive interpreter & compiler
- Forth virtual memory
- Full cursor Screen Editor
- Provision for application program distribution without licensing
- FORTH equivalent Kernal Routines
- Conditional Macro Assembler
- Meets all Forth 79 standards+
- Source screens provided
- Compatible with the book "Starting Forth" by Leo Brodie
- Access to all I/O ports RS232, IEEE, including memory & interrupts
- ROMABLE code generator
- MUSIC-EDITOR
- SPRITE-EDITOR
- Access all C-64 peripherals including 4040 drive and EPROM Programmer.
- Single disk drive backup utility
- Disk & Cassette based. Disk included
- Full disk usage—680 Sectors
- Supports all Commodore file types and Forth Virtual disk
- Access to 20K RAM underneath ROM areas
- Vectored kernal words
- TRACE facility
- DECOMPILER facility
- Full String Handling
- ASCII error messages
- FLOATING POINT MATH SIN/COS & SQRT
- Conversational user defined Commands
- Tutorial examples provided, in extensive manual
- INTERRUPT routines provide easy control of hardware timers, alarms and devices
- USER Support

SUPER FORTH 64® *is more powerful than most other computer languages!*



SUPER FORTH 64® *compiled code becomes more compact than even assembly code!*



A SUPERIOR PRODUCT in every way! At a low price of only
$96
Free Shipping in U.S.A.

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
AUTHOR INQUIRIES INVITED

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 (Established 1976)

Commodore 64 & VIC-20 TM of Commodore

Sixth FORML Conference

Forth Modification Laboratory

November 23–25, 1984

CALL FOR PAPERS

FORML is a technically advanced conference of Forth practitioners. The topics to be discussed will affect the future evolution of Forth. All conference participants are encouraged to write a paper for oral or poster presentation.

Topics Suggested for Presentation

Forth in 64K and beyond	**Forth on the 32 bit machine**
Forth expert systems	**Forth in the future**
Forth on the Macintosh	**Forth for robot control**
Forth advanced applications	**Forth programming style**

Registration and Papers

Complete the registration form, selecting accommodations desired, and send with your payment to FORML. Include a 100 word abstract of your proposed paper. Upon acceptance by FORML, a complete author's packet will be sent to you. Completed papers are due September 30, 1984.

About Asilomar

Asilomar is an ideal conference location. It is situated on the tip of the Monterey Peninsula overlooking the Pacific Ocean. Asilomar occupies 105 secluded acres of forest and dune. The secluded setting and clustered meeting and accommodation areas make it ideal for group meetings. Asilomar's excellent meals are complemented by Asilomar's homemade bread and pastries. Accommodations are excellent and deluxe rooms have been reserved for FORML attendees. Sweeping ocean views are available from decks or balconies. Asilomar is a Unit of the California State Park System.

Registration Form

Complete and return with check made out to:
FORML, P.O. Box 51351, Palo Alto, CA 94303

Name ________________________________

Company ________________________________

Address ________________________________

City ________________ State ____________ ZIP ____________

Telephone (day) ______________ (evening) ______________

I have been programming in Forth for: (years) _____ (months) _____

Accommodations Desired

Prices include coffee breaks, wine and cheese parties, use of Asilomar facilities, rooms Friday and Saturday nights, and meals from lunch Friday through lunch Sunday. Conference participants receive notebooks of papers presented.

Conference attendees, share a double room:
number of people _______ × $250 = $ ___________

Attendees in single room (limited availability):
number of people _______ × $300 = $ ___________

Non-conference guests:
number of people _______ × $200 = $ ___________

Total Enclosed $ ________

Options: Vegetarian meals? _____
Non-smoking roommate? _____

FORML, P.O. Box 51351, Palo Alto, California 94303, U.S.A.

Letters to the Editor

Kaypro User in Distress

Dear FIG:

I've read about you both in *Micro Cornucopia* and in *Rolling Stone*. (When do you expect to make *People* magazine?)

I have a Kaypro II and I'm interested in obtaining a version of Forth. I need advice to help me select from the different versions available. I've read that one version runs on CP/M and another is standalone. I've read of fig-FORTH, UNI-FORTH, Forth-79 and Forth-83. Help! Is there somebody I could call who can tell me the relative advantages and disadvantages of the various available versions?

Sincerely,

Alan Barlow
Box 3634
Seattle, Washington 98124

To Stack or Not...

Hello:

When we're learning Forth, we're told that using the stack to hold our variables is the efficient way to go. The impression I got was that the claim was valid and that there was no point of diminishing returns.

What I've found is that it seems to be a good idea to favor variables when you are working with a program that needs to access a lot of different variable values.

The listing gives an example of such a program — the version using variables runs about 2.7 times faster than the version using only the stack (the variable version sorts 800 bytes in about forty seconds on a 1.79 MHz system, versus one minute and fifty seconds with the stack version.)

Aside from being a lot easier to code for the original programmer, anyone else who wants to figure out what it does stands a better chance than the poor soul who tries to decipher the stack version.

Variables vs. the Stack

```
SCR # 52
( Shell-Metzner sort for bytes )

 0 VARIABLE M     0 VARIABLE LOC     0 VARIABLE K
 0 VARIABLE N     0 VARIABLE J       0 VARIABLE I
 0 VARIABLE L     0 VARIABLE GO
 1 CONSTANT TRUE  0 CONSTANT FALSE

: BYTE-SORT-2  ( Loc #of-bytes --- )
DUP M ! N !   1 - LOC !
BEGIN M @ 2 / DUP M ! 0 > WHILE
 N @ M @ - K !   1 J ! BEGIN
    J @ I !  FALSE GO ! BEGIN
       I @ M @ + L !
       LOC @ I @ + C@  LOC @ L @ + C@ > IF
-->

SCR # 53
          LOC @ I @ + C@  LOC @ L @ + C@
          LOC @ I @ + C!  LOC @ L @ + C!
          I @ M @ - I !
       ELSE TRUE GO ! ENDIF
    I @ 1 < GO @ OR UNTIL
  1 J +!  J @ K @ > UNTIL
 REPEAT ;

;S







SCR # 49
: BYTE-SORT ( Loc #of-bytes --- )
SWAP 1 - SWAP
DUP BEGIN 2 / DUP 0 > WHILE
  2DUP - 1 BEGIN DUP
    BEGIN DUP 5 PICK +
       7 PICK 3 PICK + C@  8 PICK 3 PICK + C@
       2DUP > IF
          9 PICK 5 PICK + C!  8 PICK 3 PICK + C!
          DROP 4 PICK - DUP 0
       ELSE 2DROP DROP DUP 1 ENDIF
    SWAP 1 < OR UNTIL
 DROP 1+ 2DUP SWAP > UNTIL
DROP DROP REPEAT DROP DROP DROP ;

( This sort works on arrays of bytes. )
(  It uses the Shell-Metzner algorithm. )
```

Editorial

Practical Forth

Sometimes we get so involved with theoretical and research aspects of programming that we forget the computing sciences (and arts) should be helping people. Not so David L. Jaffe, whose work keeps him directly involved with the humanitarian end of things daily. Find herein his report on Forth in rehabilitation applications. We can imagine no more satisfying and productive use of our technology, nor one more deserving of support.

This issue also contains a contribution to help all of you who haven't yet experienced the joy of uploading a lengthy program directly from a friend's computer instead of laboriously typing it manually. It is simple enough for newcomers to telecommunications to comprehend what it is doing (only three screens of code). Enter this one by hand and use it to upload a version of the much lengthier XMODEM when you are ready to graduate to error checking and other features.

Finally, a note to prospective authors. *Forth Dimensions* welcomes your contributions to the field of Forth literature. Our reviewers are always happy to see new material and new authors. Of particular interest are tutorials and simple applications which exemplify some feature of Forth for novices, accompanied by a clearly written, reader-friendly article. (Yes, I know that all good Forth code is self-documenting....)

Meanwhile, thanks to all of you who have taken time to write a letter or short note to tell us how we are doing. We enjoy hearing from you!

—Marlin Ouverson
Editor

We here at SEFFIG (SouthEast Florida FIG) have come to the consensus that the Forth stack is at its best when used for operations like **DUP**, **SWAP**, **OVER** and **ROT** which are very simple operations on only the first three values on the stack; I suggest that deep stacks are often more trouble than they are worth. We encourage anyone with observations, experiences or constructive criticisms of this notion to drop us a line.

Good Day!

Rudy Smith
4601 SW 58th Avenue
Miami, Florida 33155

Minus a Plus

Dear Marlin:

I was pleased to see the publication of my decompiler in *Forth Dimensions* (V/6). Unfortunately, in the process of typesetting my text, an error was introduced into the definition of **ASCII** on page eighteen; The word **1** should be **1+**.

Thanks for your continued interest.

Sincerely,

Norman L. Hills
Director of Data Processing
Servi-Share of Iowa
600 Fifth Avenue, Suite R
Des Moines, Iowa 50309

Reviews from Rochester

Dear FIG:

I am pleased to resume my subscription to *Forth Dimensions*. Marlin Ouverson and Roy Martens, together with their respective editorial and production staffs, have done a superb job with *Forth Dimensions*. Volume 5 Number 6 just confirms my thoughts! I especially enjoyed the interview with Bill Ragsdale, and hope that you will have more in the future. May I suggest that you corner Hans Nieuwenhuyzen for his thoughts? The "PL/I Data Structures" article was also good and should inspire my students to further consider the power of defining words. The montage-style cover art alternates between being confusing and illuminating, just like Forth. Finally, congratulations to John Hall for getting the chapters together and enticing them to write up their minutes. I've been lax, but I'll see that some Rochester FIG minutes (hours?) make it to *Forth Dimensions*.

Looking forward,

Lawrence P. Forsley
Laboratory for Laser Energetics
University of Rochester
Rochester, New York 14623

Getting HEXed

Dear FIG:

The number utility described by David McKibbin (*Forth Dimensions* V/4) is certainly an improvement over changing **BASE** for each number that requires it, or modifying the Forth interpreter to accept new number characteristics. I

PROGRAMMERS

Linn Electronics, makers of digital percussion synthesizers, is looking for excellent programmers to assist in the development of new products in digital percussion and other areas of computer music.

Experience is required in software development of real-time microprocessor-based systems. Knowledge of 8086 assembly code, FORTH, hardware design, and music synthesis is preferred.

Send resume and salary history to:

Linn

Linn Electronics, Inc.
Research & Development Division
18720 Oxnard Street
Tarzana, California 91356

Numeric Conversion

```
screen#     57
 0 \ Hex numeric conversion
 1 DECIMAL
 2   (HEX)    ( --- n[literal] )  \ hex convert chars at HERE 1+
 3   BASE @ >R HEX
 4   HERE 1+ CONVERT DROP         \ convert all but 1st char
 5   [COMPILE] LITERAL            \ NOTE: STATE-smart LITERAL
 6   R> BASE ! ;
 7
 8  1 WIDTH !                     \ match on 1st char and count'
 9 : $XXXX (HEX) ; IMMEDIATE      \ 4 digit number
 A : $XXX  (HEX) , IMMEDIATE      \ 3 digit number
 B   $XX   (HEX) ; IMMEDIATE      \ 2 digit number
 C · $X    (HEX) ; IMMEDIATE      \ 1 digit number
 D 31 WIDTH !
 E
 F

screen#     58
 0 \ ASCII character converters
 1 DECIMAL
 2 1 WIDTH !
 3
 4 : "X   ( --- c[literal] )
 5   HERE 2+ C@ [COMPILE] LITERAL ; IMMEDIATE
 6
 7 : ^X   ( --- c[literal] )
 8   HERE 2+ C@   "@ -           \ get char, convert to control
 9   DUP BL U< NOT 0 ?ERROR       \ legal?
 A   [COMPILE] LITERAL ; IMMEDIATE
 B
 C 31 WIDTH !
 D
 E
 F
ok
 57 LOAD ok
$100 . 256 ok
$7FFF . 32767 ok
  TEST $41 EMIT ; ok
TEST Aok
 ok
58 LOAD ok
"A . 65 ok
^A . 1 ok
$A . 10 ok
 : STARS 0 DO "* EMIT LOOP ; ok
10 STARS **********ok
```

preferred **$** to **H'** as a name for input hex conversion, more in keeping with traditional symbolism. However, if you combine this idea with the in-word parameter passing as described by Timothy Huang (*Forth Dimensions* V/3), you can get that unchanged interpreter to process numeric conversion symbols of any type (as long as they are at the head of the name field). Consider the code in screens #57-58 and the sample usages. A minor problem is that conversion failure leaves you **HEX**ed.

Sincerely yours,

Alden B. Long
23 Pleasant Avenue
South Hamilton, Massachusetts 01982

President's Letter

Goals and Objectives

The elections are complete and work has started on planning the services and activities for members during the next year. But first, a note of thanks to the founders and first Board of Directors of the Forth Interest Group. They made it possible for almost any computer enthusiast to have an outstanding language, Forth, running on his or her computer. Listings and instructional material have been available from the Forth Interest Group since 1979. Instructional meetings have been held throughout the world to demonstrate the virtues of Forth. The results of these activities are that Forth is used for many industrial groups, Forth systems are distributed by several Forth vendors, universities use Forth in course material, a Forth bulletin board system is in operation twenty-four hours a day (dial 415-538-3580; operation is currently at 300 baud), conferences and conventions devoted specifically to Forth are held regularly, and the Forth Interest Group has had a membership growth to 4713 members. This is a remarkable record for an organization directed by volunteers. Thank you Dave Boulton, Kim Harris, John James, Dave Kilbridge and Bill Ragsdale for making Forth and Forth information available to everyone.

Now a new Board of Directors has set about continuing the activities of the Forth Interest Group; they are John Hall, Kim Harris, Thea Martin, Robert Reiling and Martin Tracy. They have elected officers as follows: President, Robert Reiling; Vice President, Martin Tracy; Secretary, Kim Harris; and Treasurer, Dave Kilbridge. John Hall is the Chapter Coordinator.

The first order of business of the new Board of Directors was to set down the purpose and goals of the Forth Interest Group. Here is the list.

1. The Forth Interest Group is a non-profit, member-supported organization, which provides services to promote the use of the Forth computer language.

2. Services include education and communications to members and the public.

3. The Forth Interest Group desires that every Forth user in the world join the Forth Interest Group. To this end, the Forth Interest Group will try to service their needs.

4. The Forth Interest Group's main objective is to provide services to its members; secondary objectives are 1) to increase the membership base in order to offer better services and 2) to promote the use of the Forth computer language.

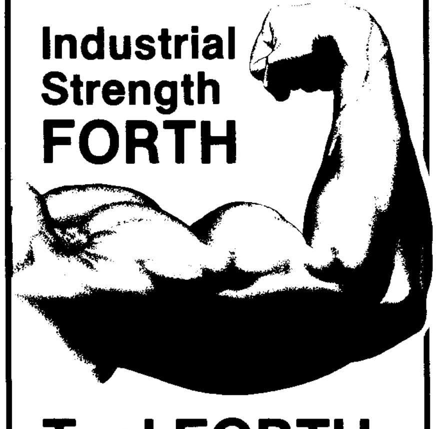
The above will be the guide followed by the Board of Directors and officers in the forthcoming operation of the Forth Interest Group.

Here are some activities already planned for this year. The FORML conference and tour program, which will be a group trip going to Taiwan, Hong Kong and China, will attend a FORML conference in Taiwan, an International Forth Interest Group dinner meeting in Hong Kong and meet with Chinese Forth enthusiasts at the Chiao Tung University in Shanghai. The group departs September 25, 1984 and returns October 14, 1984. Look in the last *FORTH Dimensions* for details of the trip. In the U.S.A., this year's FORML conference will be held at Asilomar, California, November 23, 1984 through November 25, 1984. The annual Forth Convention will be held November 16-17, 1984 at the Hyatt Palo Alto in Palo Alto, California. Look in this issue of *FORTH Dimensions* for details of the Convention and Asilomar FORML Conference.

-Robert Reiling
President

Multiuser/Multitasking
for 8080, Z80, 8086

Industrial Strength FORTH



TaskFORTH™
The First Professional Quality Full Feature FORTH System at a micro price*

LOADS OF TIME SAVING PROFESSIONAL FEATURES:

☆ Unlimited number of tasks
☆ Multiple thread dictionary, superfast compilation
☆ Novice Programmer Protection Package™
☆ Diagnostic tools, quick and simple debugging
☆ Starting FORTH, FORTH-79, FORTH-83 compatible
☆ Screen and serial editor, easy program generation
☆ Hierarchical file system with data base management

* Starter package $250. Full package $395. Single user and commercial licenses available.

If you are an experienced FORTH programmer, this is the one you have been waiting for! If you are a beginning FORTH programmer, this will get you started right, and quickly too!

Available on 8 inch disk under CP/M 2.2 or greater also various 5¼" formats and other operating systems

FULLY WARRANTIED, DOCUMENTED AND SUPPORTED

DEALER INQUIRES INVITED

Shaw Laboratories, Ltd.
24301 Southland Drive, #216
Hayward, California 94545
(415) 276-5953

Procedural Arguments

Kurt Luoto
Redwood City, California

We are all familiar with the concept referred to as "factoring" by the Forth community. It can be summarized as eliminating duplication of effort. Consider the code depicted in figure one. In it, there are three different bodies of code (A, B and C) that have some sequence of words (X) that is the same in each of them. The usual thing to do when you notice such a situation is to define a new word named **X** and to replace the sequence in each larger code portion with the word **X**. The benefits of this factoring are: 1) the code takes up less space, 2) the code is usually easier to understand, 3) we only have to debug the common code once and 4) modifications are made easier since functions are centralized. The price we pay is a small overhead for calling and returning from the new word. Everyone is familiar with this kind of factoring and examples can be found everywhere.

However, there is another kind of factoring that is often neglected. Consider the code depicted in figure two. Here there are three bodies of code that are the same in all respects except for the inner portion of each. It is a sort of dual of the first kind of factoring. Again, the thing to do here is to factor out the common code, in this case the outer portion of each, and to make it into a new word called **X**. All the same benefits and penalties apply as with the first kind of factoring.

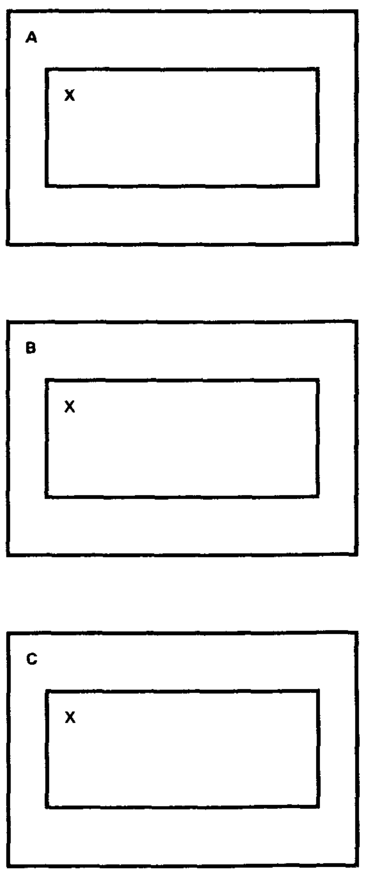


Figure One

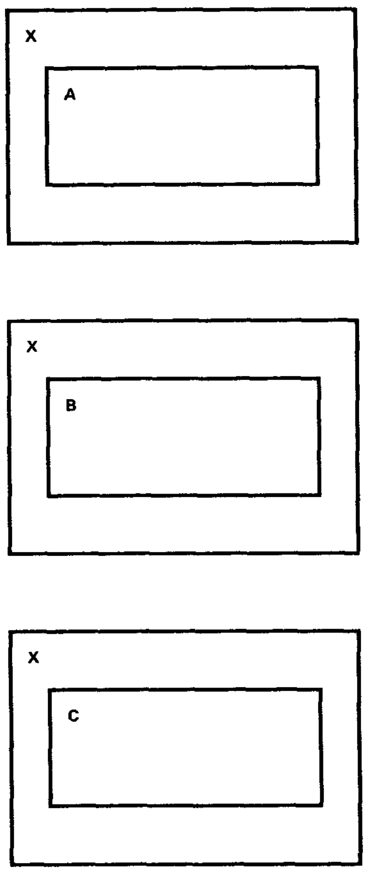


Figure Two

The extra step we must make in this case is to somehow let the new word know what code it is supposed to execute in the inner portion when it is called. One way to do this is to have a case structure in the middle and pass the word an argument indicating which case to execute. But this presumes that you know beforehand all the cases that you might want to have. It also fails to fully separate, to fully factor the word. A much nicer way is to pass not a value, but the actual code we want executed in the middle as an argument or parameter. Such arguments are called procedural or functional arguments. I will refer to structures that take procedural arguments as forms, borrowing terminology from Glass' review[1] of an article by Backus (my apologies if this usage is not precise). Let's look at a few examples.

A Simple Example

A common function in many applications is sorting. Sorting may need to be performed on a wide variety of data. While no sorting algorithm is optimal in all situations, it would still be nice to be able to write a single sorting routine that is applicable to any type of data. One way to approach this is to notice that for many sorting routines in the literature, the only portions of the algorithm that depend on the data are the comparison function, which is the way you tell whether one item is "less than" another, and the way that the logical list of items is represented.

We can write a generalized sorting routine by generalizing these two dependencies. We can generalize the latter by agreeing that our data will consist of an array of sixteen-bit words (cells) in memory. Each sixteen-bit value may be an integer, an address of some data item in memory, the index of a record in a file; in short, anything we care to represent with sixteen bits. We will pass this array to the sorting routine by giving its address and length. We take care of the first dependency by also passing the comparison function to the sorting routine as a procedural argument, i.e. the sorting routine will be a form.

Screen #182 defines a ~~word~~ **SORT** that takes three parameters on the data stack: the address of an array, the length of the array and the CFA of the comparison function. This is perhaps the most straightforward way of passing a procedure in Forth. (This sorting routine, which happens to be an insertion sort, is not a very good sorting algorithm in general. It is only efficient if there are no more than about a dozen items to be sorted, or when the array is already sorted or "nearly sorted." It is used here only as an example, Those wishing more information on sorting may look in any number of references[2].)

Thus, if we had an array **A** of (signed) integer values of length **LEN**, then the sequence **A LEN ' < PFA>CFA SORT** would sort the array in ascending order. The sequence **A LEN ' PFA>CFA SORT** would sort the array in descending order. If we wanted to treat the values as unsigned integers, then we would use **A LEN ' U< PFA>CFA SORT** Suppose that the array of values is actually a list of addresses of strings that we wish to sort alphabetically. Then we would define a word like **$<** that would take two string addresses and return true if the first string was "less than" (alphabetically before) the second and false otherwise. Again, we would use **A LEN ' $< PFA >CFA SORT** to sort the list in alphabetical order.

By now you can see how to sort almost anything in this manner. We could even use a different sorting algorithm, as long as it took the same parameters, and it would not require any other changes in the code that used the sort. Such are the advantages of good factoring. Of course, faster sorts can be obtained by tailoring them to specific situations, but that is the price we pay for generality.

Many kinds of forms can be described by sentences such as, "On every x of a given structure y, perform F," where F is a procedural argument. One example is, "On every word x in a given vocabulary, perform F," where F might be, "Print the name of x," or "Print whether x is a **CONSTANT**." Another is, "On every leaf x of a given binary tree, visited in preorder, perform F," where F might be, "Append the data associated with leaf x to a list in memory." This last one might be implemented in a recursive fashion. Sometimes we may want to create new forms in terms of previously defined ones. For example, the form "On every file of type y on the disk, perform F" might be combined with the form "On every record x of a given file, perform F" to make a new form, described as "On every record x of every file of type y on the disk, perform F." Many more examples of these and other types of forms can be found and these make good candidates for factoring.

Other Methods of Passing Procedural Arguments

While passing procedural arguments by putting their CFAs on the data stack is straightforward and sufficient for some forms, it can become quite messy when many parameters must be manipulated within the form or when several procedural arguments must be passed. Therefore, it is good to consider other methods also.

In Laxen[3] and Bilobran[4], procedural arguments are passed in variables. These variables are referred to as "execution vectors." This method is especially appropriate where modal behavior of a word is needed, i.e. a word operates in a certain way each time it is called until some execution vector is changed. Execution vectors are rather limited in application, though. We will not be looking at these much more in this article.

In McKibbin[5] and Laxen[6], procedural arguments (CFAs) are placed in the parameter portion of a word created with a **CREATE DOES>** construct. In addition, Laxen discusses the idea of a stack on which to pass procedural arguments. This is a very useful idea, since sometimes just the two stacks (data and return) are not enough to conveniently handle all parameters. But while Laxen's example is instructive, it also has its limitations. So let's take a look at a more general approach to making and using forms.

First, let's list some of the properties that we want in form handling.

1) We want the ability to combine forms, that is, to create new forms that call upon previously defined ones.

2) We want our forms to be re-entrant and we want to be able to make them recursive.

3) We want to be able to handle forms that take a variable number of procedural arguments.

4) We want our defining words to do as much of the bookkeeping as possible, i.e. make the forms as simple as possible to use.

In the accompanying screens are some definitions that I think meet these criteria. Let me show you how they are used and see if you agree.

A form is defined with the word pair **FORM END-FORM**. They are used in the same way as : and ;, the format being **FORM** <name of form> <body> **END-FORM** where <body> is just like a colon definition body. Within the body of a form, a procedure that has been passed as an argument can be executed by the sequence <argument number> **%EXEC** where <argument number> is zero for the first argument passed, one for the second, etc. The word **%NARGS** places on the data stack the number of (the count of) arguments that have been passed to the form.

A call to a form has the format <arg 0><arg 1><arg 2>...<arg n><name of form> where <arg x> is a construct that pushes a procedural argument onto a special stack called the procedure stack. A form may be called with any number of arguments, including none at all. There are several formats for pushing a procedural argument on the stack, including the following:

1) An argument may have the format <CFA> **%PUSH** where <CFA> is the CFA of a word (procedure) to be passed. For example, you could use ' <word name> **%PUSH** to pass a particular word as an argument.

2) Within a form, an argument may have the format <argument number> **%ARG**

where <argument number> is the number of an argument (procedure) that has been passed to the form.

3) An argument may have the format **%:** <body> **%;** where <body> is any body of code as would normally appear in a colon definition, except for the definition name. A headerless piece of code is generated in-line and is passed as an argument.

4) An argument may have the format **%CODE** <body> **%END-CODE** where <body> is any body of code as would normally appear in a **CODE** definition, except for the definition name. A headerless piece of code is generated in-line and is passed as an argument.

Recursion may also be used in forms. In place of

<arg 0><arg 1><arg 2>...<arg n><name of form>

you may use

<arg 0><arg 1><arg 2>...<arg n> **MYSELF**

where you wish to recursively call the form that you are defining. The word **MYSELF** is the (almost) standard Forth word which compiles the CFA of the word currently being defined. If, however, you are going to call a form recursively with exactly the same list of procedural arguments that it was called with, you may use **%MYSELF** which is more efficient (it does not take up extra space for the call and takes less time). The word **%MYSELF** can only be used within a form definition, i.e. between **FORM** and **END-FORM**.

Given these words, defined in screens #183-190, we can now use forms in a fairly general manner. Let's look at a few examples.

Screen #195 re-defines our sort in terms of the new defining words. The word **ISORT** which sorts an array of integer values is defined using the new form.

Screen #197 defines a form called **PER-WORD** that takes the address of a header of a word, usually the first one in a vocabulary, and passes it to the single procedural argument that was passed to it. After calling the procedural argument, it expects a flag on the data stack indicating whether to continue or not. If

```
   PROCEDURAL ARGUMENTS

BLOCK: 182

 ( SORTING FORM                               VLFORTH    KWL 01JAN84   )

 : SORT  ( ADDR  LEN  CFA -- )    ( INSERTION SORT )
    ( SORTS TABLE AT GIVEN ADDRESS WITH GIVEN LENGTH USING )
    ( THE GIVEN ROUTINE FOR THE COMPARISON OPERATOR          )
    ROT ROT   2 - CELLS   OVER +   SWAP OVER
    DO   I   DUP @   SWAP   NEXT-CELL @   SWAP 4 PICK EXECUTE
       IF   I @   I
          BEGIN   DUP NEXT-CELL @   OVER !   NEXT-CELL
             OVER   OVER NEXT-CELL @   SWAP 6 PICK EXECUTE
             OVER  5 PICK  > 0=   AND 0=
          UNTIL   '
       THEN
    -1 CELLS +LOOP   DROP DROP   ;

 : ISORT  ' < PFA>CFA   SORT  ;


BLOCK: 183

 ( HELPING WORDS                              VLFORTH    KWL 01JAN84   )
 FORTH DEFINITIONS   DECIMAL
 ( THE WORD 'COLON' COMPILES THE RUNTIME CODE FOR COLON DEFNS. )
 ( THUS YOU SEE IT COMPILED IN AT VARIOUS POINTS WHERE A CFA   )
 ( IS TO BE LOCATED.                                           )
 : COLON   [ LATEST LFA>CFA @ ] LITERAL ,   ;

 ( THE CONSTANT 'CELL' INDICATES THE NUMBER OF MACHINE ADDRESS )
 ( UNITS THAT MAKE ONE CELL, I.E. ONE 16-BIT WORD IN 16-BIT    )
 ( IMPLEMENTATIONS OF FORTH.  ON A TYPICAL 8-BIT MACHINE, THIS )
 ( CONSTANT WILL BE TWO, SINCE TWO BYTES MAKE A 16-BIT WORD.   )
 ( HOWEVER, THIS IMPLEMENTATION USES WORD ADDRESSES.           )

 1 CONSTANT CELL
 : CELLS  ( N -- OFFSET )    CELL *   ;
 : NEXT-CELL   ( ADDR -- ADDR-PLUS-ONE-CELL )  1 CELLS +  ;    -->


BLOCK: 184

 ( FORM DEFINING WORDS                        VLFORTH    KWL 01JAN84   )
 FORTH DEFINITIONS   DECIMAL

 VARIABLE  %FRAME    ( FRAME POINTER )
 VARIABLE %STACK  200 ALLOT  ( FIRST WORD OF STACK IS POINTER )

 : %STACK!   ( -- )  ( RESETS STACK AND FRAME POINTERS )
   %STACK   DUP NEXT-CELL SWAP   DUP %FRAME !   !   ;

 %STACK!

 ( NOTE: %STACK!  SHOULD BE COMPILED INTO ABORT CODE )

 -->

BLOCK: 185

 ( FORM DEFINING WORDS                        VLFORTH    KWL 01JAN84   )

 : (%PUSH)  ( N -- )  ( PUSHES A VALUE FROM THE DATA STACK )
                      ( ONTO THE PROCEDURE STACK           )
   %STACK @ !   1 CELLS %STACK +!   ;

 : (%POP)   ( -- N )  ( POPS A VALUE FROM THE PROCEDURE    )
                      ( STACK ONTO THE DATA STACK          )
   -1 CELLS %STACK +!   %STACK @ @   ;

 : %PUSH    ( CFA -- )
   ( PUSHES A CFA AND THE CURRENT FRAME VALUE ONTO THE STACK )
   (%PUSH)   %FRAME @ @ (%PUSH)   ;

 : %'   ( CELL FOLLOWING THIS HAS CFA TO PUSH ONTO PROC. STACK )
      R>   DUP NEXT-CELL >R   @ %PUSH   ;              -->
```

```
BLOCK: 186

 ( FORM DEFINING WORDS                        VLFORTH    KWL 01JAN84   )

 : (%:)    ( -- )
   ( THE CELL FOLLOWING THIS HAS ADDRESS TO BRANCH TO. )
   ( THE CELL FOLLOWING THAT IS THE CFA OF THE INLINE  )
   ( PROC. ARG WHICH WILL BE PUSHED ON THE PROC. STACK.)
   R>   DUP NEXT-CELL    SWAP @ >R    %PUSH    ;

 : %:    ( INTRODUCES AN INLINE PROC. ARG. COLON DEFINITION )
   ?COMP    COMPILE (%:)    HERE 0 ,    COLON    31    ;
   IMMEDIATE

 : %;    ( CLOSES AN INLINE PROC. ARG. COLON DEFINITION )
   ?COMP    31 ?PAIRS    COMPILE EXIT    HERE SWAP !    ;
   IMMEDIATE
 -->

BLOCK: 187

 ( FORM DEFINING WORDS                        VLFORTH    KWL 01JAN84   )

 : (FORM)       ( CELL FOLLOWING IS CFA OF FORM TO BE CALLED )
   0 (%PUSH)   ( TO INDICATE END OF ARGUMENT LIST. )
   ( PUSH NEW FRAME POINTER )
   %STACK @    %FRAME @ NEXT-CELL (%PUSH)    %FRAME !
   R>  EXECUTE   ( EXECUTE THE FRAME CODE )
   ( RESTORE FRAME POINTER AND POP FRAME OFF OF STACK )
   (%POP)    DUP %STACK !    -1 CELLS +    %FRAME !    ;

 -->

BLOCK: 188

 ( FORM DEFINING WORDS                        VLFORTH    KWL 01JAN84   )
 : FORM    ( FORM DEFINING WORD. USED IN PLACE OF : )
    [COMPILE] :    COMPILE (FORM)    COLON    30    ;     IMMEDIATE

 : END-FORM    ( FORM DEFINIG WORD. USED IN PLACE OF ; )
    30 ?PAIRS    [COMPILE] ;    ;     IMMEDIATE

 : %NARGS    ( -- N )
    ( PUTS NUMBER OF PROC. ARGS. IN THE CURRENT FRAME )
    ( ON THE DATA STACK                               )
    0    %FRAME @ @
    BEGIN    DUP @    WHILE
        2 CELLS +      SWAP 1+ SWAP    REPEAT    DROP    ;
 -->

BLOCK: 189

 ( FORM DEFINING WORDS                        VLFORTH    KWL 01JAN84   )

 : (ARG)    ( N -- ADDR )
    ( CALCULATES ADDRESS OF THE PROC. ARG. ON THE STACK )
    2 CELLS *    %FRAME @ @    +    ;

 : %EXEC    ( N -- )    ( EXECUTES PROC. ARG. N )
    ( GET ADDR. OF ARGUMENT.  SAVE CURRENT FRAME AND    )
    ( SET UP FRAME ASSOCIATED WITH THE ARGUMENT         )
    (ARG)    %FRAME @ (%PUSH)
    %STACK @    OVER NEXT-CELL @    (%PUSH)    %FRAME !
    @ EXECUTE    ( EXECUTE THE ARGUMENT )
    ( RESTORE THE STACK AND FRAME POINTERS )
    (%POP) DROP    (%POP) %FRAME !    ;

 -->
```

the flag is false, the form returns. If the flag is true, the form gets the next word from the LFA of the current word and again passes it to the procedural argument, repeating until a false flag is left on the stack. This word is dependent on the structure of the header and vocabulary in the system.

The word **VLIST** is defined using this form. It lists the names of all the words in a vocabulary (or vocabularies).Note that **PER-WORD** keeps its copy of the word address of the return stack while calling its procedural argument. This is to make it a little easier for procedural arguments to use the data stack. For example, **WORD-STATS** is defined to count the number of words in the linked list, calculate the sum of the lengths of their names and print the average length. The word **FIND-CONSTANT** searches the list for a constant that has the given value and returns its address and true if found, false if not found. A high-level definition of **FIND** could also be written using this form.

Screens #191-192 build a contrived example of a binary tree, each node having a word of text as its data. The word **.NODE** prints the word (data) associated with a node, given its address. Screen #193 then defines forms that visit the nodes of the tree in in-order and preorder fashion respectively. These are examples of recursive forms.

Sometimes procedural arguments can be used to solve the forward referencing problem. For example, Sommers[7] gives an example of several routines that call each other recursively. The problem is that the definitions would normally require some forward referencing. Sommers shows how this can be solved using execution vectors. His code is reproduced here in screens #199-200, modified slightly for this version of Forth.

With the form-handling words presented here, another approach is possible. Equivalent code using these words is given in screen #201. (Here I have eliminated unnecessary variables.) In this implementation it should be noted that within the definition of a form, any occurrence of the phrase <argument number> **%EXEC** will execute an argument to the form being defined, even if this phrase is executed by another form.

For example, suppose we have the definitions

FORM B 0 %EXEC END-FORM
and
FORM A %: 0 %EXEC %; B END-FORM

When **A** is called, it calls **B** with the indicated in-line code argument. **B** in turn calls the argument which executes the phrase **0 %EXEC**. Since the context of this phrase was **A**, the first argument that was passed to **A**, not **B**, is executed even though the form currently executing is **B**. That is, arguments "remember" their context. This is somewhat similar to the funarg problem in LISP. It may sound odd when explained like this, but it is very natural in use so you don't usually need to think about it.

Implementation Notes

The procedure stack is simply a user-defined stack on which to store CFAs and other information (see figure three). In my implementation, the stack grows upward towards high memory. There is an additional variable called the frame pointer that points to the current argument list (actually it points to a pointer to the argument list).

In a typical call to a form, the various procedural argument constructs push a CFA onto the stack along with the value of the frame pointer that was in effect when that procedure was first pushed onto the stack. This is so we can restore the "context" of that routine when we go to execute it.

A special run-time word, **(FORM)**, is used to actually call the body of the form. It pushes a zero on the procedure stack (to mark the end of the argument list), then pushes the current value of the frame pointer and sets the frame pointer to the new frame. Next, the run-time code of the form is called by **(FORM)**. When the form returns, **(FORM)** restores the previous frame pointer and the entire frame is popped off the procedure stack.

When a procedural argument is executed, the current frame pointer is saved on the procedure stack (the return stack could be used instead if it has enough room), the frame value associated with the argument is stored in the frame pointer and the procedure is executed. When

```
BLOCK: 190

( FORM DEFINING WORDS                        VLFORTH   KWL 01JAN84  )

: %ARG     ( N -- )    ( PUSHES PROC. ARG. N ONTO THE PROC. STACK )
  (ARG)   DUP @ (%PUSH)    NEXT-CELL @ (%PUSH)    ;


: %MYSELF      ( CALLS THE CURRENT FORM RECURSIVELY, SAME ARGS. )
  ?COMP   LATEST LFA>PFA NEXT-CELL   ,   ;    IMMEDIATE


BLOCK: 191

( TREE-FORMING WORDS                         VLFORTH   KWL 01JAN84  )

: NODE   ( ADDR -- ADDR' )
  HERE SWAP !    HERE    0 , 0 ,     ( LEFT & RIGHT POINTERS )
  BL WORD    C@ 1+ ALLOT      ( WORD OF TEXT FOR DATA )     ;

: SONS   ( ADDR -- ADDR ADDR' )
  DUP NEXT-CELL  ;

: ENDSONS    ( ADDR -- )       DROP  ;

: .NODE     ( ADDR   -- )    2 CELLS +    .ID  SPACE    ;

-->



BLOCK: 192

( TREE EXAMPLE                               VLFORTH   KWL 01JAN84  )

VARIABLE TREE  TREE
  NODE IS    SONS
     NODE THIS  ENDSONS
  NODE IN    SONS
     NODE TREE    SONS
        NODE A ENDSONS
     NODE PRINTED ENDSONS
  NODE INORDER
  NODE USING SONS
     NODE   FASHION ENDSONS
  NODE FORMS.    DROP

-->


BLOCK: 193

( TREE-PRINTNG WORDS                         VLFORTH   KWL 01JAN84  )

FORM  INORDER    ( ADDR -- )
   BEGIN   DUP   WHILE
      DUP NEXT-CELL @   %MYSELF      DUP  0 %EXEC
     @  REPEAT    DROP
END-FORM
: .INORDER    ( -- )    CR    TREE @   %' .NODE   INORDER    CR    ;

FORM  PREORDER   ( ADDR -- )
   BEGIN  DUP WHILE
      DUP  CR   0 %EXEC   DUP NEXT-CELL  @
      %: 3 SPACES   0 %EXEC %;   MYSELF
     @  REPEAT    DROP
END-FORM
: .PREORDER    ( -- )    CR   TREE @    %' .NODE   PREORDER    CR  ;
```

```
BLOCK: 195

 ( SORTING FORM                                VLFORTH   KWL 01JAN84   )

 FORM SORT   ( ADDR  LEN -- )    ( INSERTION SORT )
    ( SORTS TABLE AT GIVEN ADDRESS WITH GIVEN LENGTH )
    2 - CELLS    OVER +    SWAP OVER
    DO    I    DUP @    SWAP    NEXT-CELL @    SWAP 0 %EXEC
       IF    I @    I
          BEGIN    DUP NEXT-CELL @    OVER !    NEXT-CELL
             OVER    OVER NEXT-CELL @    SWAP 0 %EXEC
             OVER  5 PICK  > 0=    AND 0=
          UNTIL    !
       THEN
    -1 CELLS +LOOP    DROP
 END-FORM

 : ISORT    %' <   SORT  ;

BLOCK: 196

 ( SORTING EXAMPLE                             VLFORTH   KWL 01JAN84   )

 HEX
 VARIABLE A    -1 CELLS ALLOT
  67 , 14 , 20 ,  7 ,  9 ,      11 , 33 ,  8 , 14 , 15 ,
  33 , 22 ,  7 , 20 , 54 ,      32 ,  2 , 15 , 22 ,  7 ,

  A  14  ISORT    CR   A 14 DUMP    CR

 ( HEX RESULTS PRINTED SHOULD BE:                             )
 (  2 ,  7 ,  7 ,  7 ,  8 ,       9 , 11 , 14 , 14 , 15 , )
 ( 15 , 20 , 20 , 22 , 22 ,      32 , 33 , 33 , 54 , 67 , )

 DECIMAL



BLOCK: 197

 ( VOCABULARY FORM                             VLFORTH   KWL 01JAN84   )
 FORM  PER-WORD     ( LFA   --   )
    >R    BEGIN    R   0 %EXEC    WHILE    R> @ >R    REPEAT    R> DROP
 END-FORM

 : VLIST    ( -- )    ( PRINT NAMES OF WORDS )
    CR  LATEST  %:   ?WAIT     DUP IF    LFA>NFA .ID
                              SPACE    1 THEN    %:    PER-WORD    ;

 : FIND-CONSTANT   ( N -- [PFA] FLAG )
    ( FIND A CONSTANT WITH THE GIVEN VALUE )    LATEST
    %:   DUP IF   DUP LFA>CFA @    [ ' 0 PFA>CFA @ ]  LITERAL =
          IF    OVER    OVER LFA>PFA @   =
             IF   SWAP DROP    LFA>PFA  1   0
             ELSE    DROP   1 THEN       ELSE    DROP    1 THEN
     ELSE   DROP DROP 0 0   THEN    %;    PER-WORD    ; -->

BLOCK: 198

 ( VOCABULARY FORM                             VLFORTH   KWL 01JAN84   )
 : .CONSTANT    ( N -- )   ( USES FIND-CONSTANT )
    DUP   FIND-CONSTANT    CR
    IF   PFA>NFA   .ID   ELSE   ." NO CONSTANT" THEN
    ."   HAS VALUE " .    CR    ;


 HEX
 : WORD-STATS    ( -- )   ( VOCABULARY STATISTICS )
    0 0    LATEST
    %: DUP IF    NFA C@    1F AND    +    SWAP 1+ SWAP    1 THEN %;
    PER-WORD
    CR ." NUMBER OF WORDS: "    OVER .
    CR ." TOTAL OF NAMES LENGTHS: "    DUP .
    CR ." AVERAGE LENGTH: "    SWAP /    .    ;
 DECIMAL
```

it returns, the frame pointer is restored to its original value.

The word **%MYSELF** — rather than using **(FORM)** — simply compiles in the CFA of the run-time code of the form currently being defined. The procedure stack and argument list are left untouched.

In actual practice, all of the run-time words — **(FORM)**, **%PUSH**, etc. — should be re-defined as **CODE** words (assembly language) for efficiency. The definitions presented here model what should happen. Also, **ABORT** should reset the pointer for the procedure stack as well as for the system stacks.

I also find it is handy to have words to shorten certain common sequences, for example **%'** <word> to take the place of **'** <word> **PFA>CFA %PUSH**. Some of these could also be **CODE** words.

Note to 6502 users: Some versions of Forth on the 6502 do not allow a code field to straddle a page boundary. In this case you must take care, since the definitions of **FORM END-FORM** and **%: %;** presented here could put code fields on page boundaries. The simplest way I have found on my own system is to redefine **EXECUTE** to accept CFAs on page boundaries and then make sure that CFAs defined by these words are always executed with this new version of **EXECUTE**. This includes having an extra run-time word for use by **%MYSELF** that executes the word (CFA) following it as an in-line argument.

Summary

I have shown one technique for handling forms and procedural arguments. This is by no means the only one and probably not even the best. But I hope it causes the average Forth programmer to give more thought to the uses of procedural arguments to produce good factoring.

Acknowledgments

My special thanks to ROLM Corporation for use of their facilities in preparing this article. Also to Gerald Gutt for supplying me with VLFORTH, which the screens are written in.

References

1. Glass, Harvey, "Functional Programming in Forth," *Forth Dimensions* III/5, pg. 137.

2. Knuth, Donald E., *The Art of Computer Programming*, Vol. 3, "Sorting and Searching," Addison-Wesley, Reading, Massachusetts, 1973.

3. Laxen, Henry, "A Techniques Tutorial: Execution Vectors," *Forth Dimensions* III/1, pg. 174.

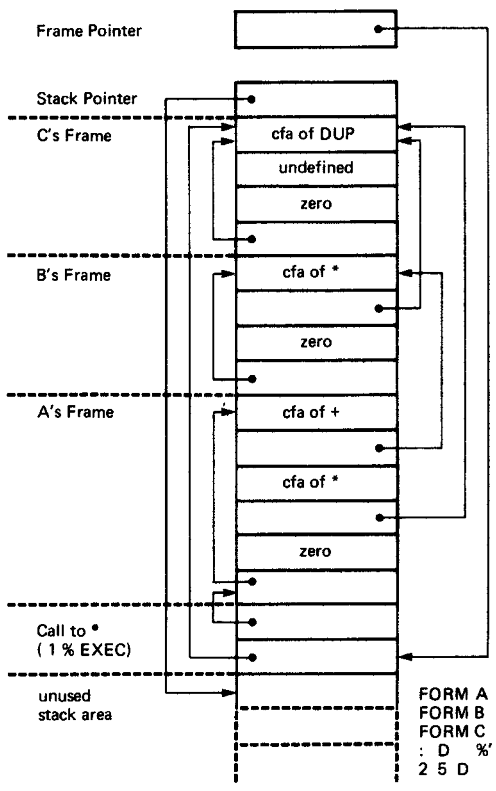
4. Bilobran, Bill, "Execution Variables," *1981 FORML Conference Proceedings*, Vol. 1, pp. 245-255, Asilomar, California.

5. McKibbin, David, "Parameter Passing to DOES>," *Forth Dimensions* III/1, pg.14.

6. Laxen, Henry, "A Techniques Tutorial: Parameterized CREATE DOES>," *Forth Dimensions* IV/5, pg. 27.

7. Sommers, Roy W., "Vectored Execution and Recursion," *Forth Dimensions* V/4, pg. 17.

```
BLOCK: 199

( SOMMERS' RECURSIVE TREE              VLFORTH    KWL 01JAN84   )
VARIABLE 'LBR      VARIABLE 'RBR      ( USED TO STORE PFAS )
VARIABLE LEVEL    8 LEVEL !     ( VALUE OF CALLING LEVEL )
VARIABLE ANGLE   20 ANGLE !     ( ANGLE BETWEEN STEMS      )

: RETURN     R> DROP    ;   ( RETURNS TO CALLING LEVEL )
: LBR    'LBR @ PFA>CFA EXECUTE    ;    ( DEFINE IN TERMS OF 'LBR)
: RBR    'RBR @ PFA>CFA EXECUTE    ;    ( DEFINE IN TERMS OF 'RBR)

: NODE
   LEVEL @    1 <    IF RETURN THEN
   -1 LEVEL +!    ANGLE @ LEFT    ( ADJUST LEVEL AND TURN )
   LBR          ( DRAW LEFT BRANCH )
   ANGLE @ 2* RIGHT          ( TURN )
   RBR          ( DRAW RIGHT BRANCH )
   ANGLE @ LEFT    1 LEVEL +!    ;    ( RESET )        -->

BLOCK: 200

( SOMMERS' RECURSIVE TREE              VLFORTH    KWL 01JAN84   )
: (LBR)
   DUP 2* PENDOWN FORWARD          ( DRAW 2X STEM )
   NODE                            ( DO NEXT LEVEL )
   DUP 2* PENUP    BACKWARD    ;   ( RESET CURSOR )

: (RBR)
   DUP   PENDOWN FORWARD         ( DRAW 1X STEM )
   NODE                          ( DO NEXT LEVEL )
   DUP   PENUP    BACKWARD   ;   ( RESET CURSOR )

' (LBR) 'LBR !    ( STORE PFA IN 'LBR )
' (RBR) 'RBR !    ( STORE PFA IN 'RBR )

: SETUP    8 LEVEL !    10 LBR    DROP    ;
( EXECUTE SETUP TO DRAW TREE )


BLOCK: 201

( SAME TREE USING FORMS                VLFORTH    KWL 01JAN84   )
: NOOP   ;

FORM  BR    ( LEN ANGLE LEVEL -- LEN ANGLE LEVEL )
   3 PICK    0 %EXEC    PENDOWN    FORWARD
   1 %EXEC
   3 PICK    0 %EXEC    PENUP      BACKWARD      END-FORM

: NODE     ( LEN ANGLE LEVEL   --   LEN ANGLE LEVEL )
   DUP  IF
      -1 +    OVER LEFT      %' 2*     %' MYSELF  BR
          OVER 2* RIGHT     %' NOOP   %' MYSELF  BR
      OVER LEFT    1+  THEN    ;

: SETUP    10 20 8    %' 2*   %' NODE   BR    DROP DROP DROP ;

```



```
FORM A   1 %EXEC    ENDFORM
FORM B   %' +   0 %ARG   A   ENDFORM
FORM C   %' *   B   0   %EXEC   ENDFORM
: D   %'   DUP   C ;
2 5 D
```

The above code yields the procedure stack shown at the left at the point that the form A calls *

Figure Three

FOR TRS-80 MODELS 1, 3 & 4
IBM PC, XT, AND COMPAQ

WHICH ONE?

Which microcomputer word processor lets you create and edit without retyping, but *won't* slow down your creative process? Knows when to capitalize the first letter while replacing one phrase with another? Can outdent as well as indent? Will do typesetting at your command, even with proportional characters, right justification and tabbed columns? Lets you use the same (extra-capacity) data disks on IBM PC *and* TRS-80? And eases your learning with common-sense keystrokes, Help menus, good examples and a professionally authored manual?

Hint: it can integrate to communicate from home to office, and will interface with a database for form letters, data tables, and more!

It's the professional's word processor for your IBM PC, Compaq, or TRS-80 Model 1, 3 or 4:

FORTHWRITE
in
MMSFORTH

With an unusually powerful set of tools and an unusually easy way of helping you to use them.

All necessary software (FORTHWRITE and the MMSFORTH System), under Personal License for $425 (IBM PC). Or with FORTHCOM telecommunications module and the DATAHANDLER-PLUS files management system, for $550 complete. Corporate Site (multiple users) License is available.

May we tell you more?

MILLER MICROCOMPUTER SERVICES
61 Lake Shore Road, Natick, MA 01760
(617) 653-6136

The Integer Solution

Marc Perkel
Springfield, Missouri

The title of this article is taken from "The TO Solution" by Paul Bartholdi (*Forth Dimensions*, I/4,5). Dealing with this subject in a different way, I chose to use the name **INTEGER** purely for psychological reasons, so as to put this concept in a class by itself. To use the name **VARIABLE** here might imply that I intend to replace the present concept of **VARIABLE**, which is not what I want to do. Not only that, the name **INTEGER** implies that I'm talking about sixteen-bit words and not strings, stacks, arrays or floating-point numbers.

Integers are self-fetching variables. Normally, they act like constants but, when following the operator -> ("to") they store the stack top. In other words, **INTEGER X** creates **X** with initial value zero. When **X** executes, it leaves its value on the stack. When the phrase **5** -> X executes, a five is stored in **X**.

Some of the reasons for usings **INTEGER**s include the symmetry between them and constants, not having to use @ and ! and making Forth programs significantly more readable. How much more readable wasn't apparent until I changed my new compiler to use integers. It creates a whole new style of programming. Consider the following task: I want to write a word **MORE** that will load the next several screens after the one I'm presently on. For example, **3 MORE** loads the next three screens. The ordinary way to code this is shown in figure one-a, and the method with integers is shown in figure one-b.

```
: MORE ( n -  | load n more screens )
    BLK @ 1+ DUP ROT + SWAP DO I LOAD LOOP ;
```

Figure One-A

```
: MORE ( n -  | load n more screens )
    BLK + 1+ BLK 1+ DO I LOAD LOOP ;
```

Figure One-B

And consider the comparison in figure two-a and two-b. In this example, the word **BOX** comes in with the length, width and height on the stack, and leaves with the volume, surface area and length of edges on the stack.

```
: BOX ( length width height - volume surface edges )
    >R 2DUP R@ * * R> SWAP >R
    >R 2DUP * SWAP ROT
    DUP  R@ * SWAP ROT
    DUP  R@ * SWAP ROT
    R> + + 4 * >R
    + + 2* R> R> ROT SWAP ;
```

Figure Two-A

```
INTEGER LENGHT    INTEGER WIDTH     INTEGER HEIGHT

: BOX ( length width height - volume surface edges )
    -> HEIGHT  -> WIDTH  -> LENGHT
    LENGHT WIDTH HEIGHT * *
    LENGHT WIDTH * LENGHT HEIGHT * WIDTH HEIGHT * + + 2*
    LENTHT WIDTH HEIGHT + + 4 * ;
```

Figure Two-B

As you can see from these examples, there is a more far-reaching impact on programming style than merely eliminating the need for @ and !. The use of integers creates an environment that inspires the writing of very readable code, especially in the area of stack manipulation, which is one of the major complaints about Forth.

Now we'll deal with the implementation techniques of **INTEGER**. There are several good ways to go about this. The best one so far, which was suggested to me by Henry Laxen, (thanks, Henry) is like **CONSTANT** with a second code field on the end. The second code field is for writing to the integer. (See figure three.)

In this implementation, the read code field and the stored integer are exactly the same as what **CONSTANT** compiles. The addition is the write code field which points to native machine code that causes the top of the stack to be stored into the integer.

To define some terms here, I will refer to the address of the read code field as the *read address*, the address where the integer is stored as the *integer address* and the address of the write code field as the *write address*.

In the normal course of interpretation, the read address is what is normally compiled or executed. This works exactly like **CONSTANT**. The write operator -> can be defined as in figure four.

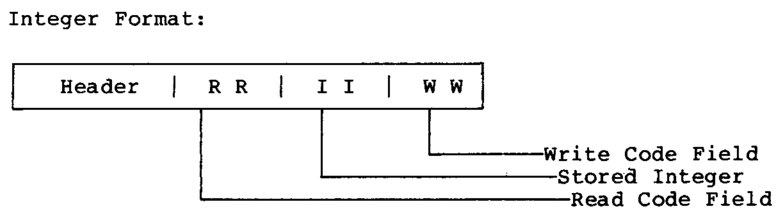


Figure Three

```
INTEGER DUMMY   FIND DUMMY 4 + CONSTANT >VAR

: -> ( integer write operator )
     FIND ?DUP NOT IF HERE ID. ." Say What?" ABORT THEN
     4 + DUP @ >VAR = NOT IF HERE ID. ." Ain't an Integer"
     ABORT THEN STATE @ IF , ELSE EXECUTE THEN ; IMMEDIATE
```

Figure Four

```
: [+>] ( adds top of stack to the address which follows )
     R> DUP 2+ >R @ +! ;

: +> ( compiles [+>] )
     STATE @ IF COMPILE [+>] FIND 2+ ,
     ELSE FIND 2+ +! THEN ; IMMEDIATE
```

Figure Five

```
: PRINT-BASE ( prints current base - assumes integers )
     SAVE BASE  DECIMAL BASE .  RESTORE BASE ;

: RECREATE ( creates new definition of following word and
                returns the address of the old definition )
     SAVE >IN  FIND  RESTORE >IN  CREATE ;
```

Figure Six

Of course, if **STATE** were an integer, the @ would be eliminated. At this point, we see that both the read and write operators compile only two bytes instead of four. This saves two bytes for every reference to variables. Not only that, both reads and writes are faster.

The definition of -> uses the 79-Standard **FIND** which returns the CFA or zero. Error checking is provided in two ways. First, the next word after -> is checked to see that it exists. Then, if it exists, it is verified as being an integer. After these checks are made, the address is incremented by four to point to the write address. Thus, the word behaves the same whether you are compiling or executing.

The concept of integers is powerful in itself but, as a side effect, it generated a series of other words to expand its usefulness even further. Along with -> consider four additional definitions: +>, **SAVE**, **REPLACE** and **RESTORE**.

As you can probably guess, +> is the integer equivalent of +!. It adds the top of stack to the integer. In the explanation of these words, I'm using high-level Forth definitions. These can easily be changed to native code for optimum performance.

We can define +> as in figure five. Note that we are dealing with the integer address itself here. +> is state smart so as to work the same in either the compile or execution state. In this example, error checking was left out but could have been included.

The next word **SAVE** saves the following integer on the return stack. That is, **SAVE DUMMY** saves the value of **DUMMY**. It replaces the phrase **DUMMY @ >R**. Used in conjunction with **SAVE** is **RESTORE** which does the opposite of **SAVE**. **RESTORE DUMMY** is equivalent to **R> DUMMY !**. The word **REPLACE** is similar to **SAVE** but also stores a new value into the integer. **REPLACE DUMMY** is equivalent to **DUMMY @ >R DUMMY !**.

Since **SAVE**, **RESTORE** and **REPLACE** are best implemented in native code, and the technique would be similar to that for +>, I am not going to code these words here. Some examples of usage are shown in figure six.

Continued on page 35

Forth Control Structures

David W. Harralson
Yorba Linda, California

The Forth machine is a hypothetical stack computer that executes instructions sequentially. In order to implement changes in program flow, a branching capability has been provided. There are two branch types defined in the Forth-83 version of the Forth virtual machine: **BRANCH** for unconditional branches, and **?BRANCH** for branching on a true stack condition.

The high-level elements of the Forth-83 language have defined the following control structures for controlling program flow:

IF... (ELSE)... THEN
BEGIN ...WHILE... REPEAT
BEGIN ...UNTIL
DO... (LEAVE) ... (+)LOOP

In this tutorial, we will take a look at the Forth machine and its control flow, and at the high-level Forth language to see how high-level control statements are translated into the Forth machine language. We will also take a look at how we can othogonalize and generalize Forth control structures while retaining compatibility with the current Forth-83 Standard.

For each word we discuss, a definition much as in the form used in the Forth-83 Standard will be given. Sample implementation code will be given, but with all the code presented separately. The sample implementation is given for an 8080 processor using absolute sixteen-bit addresses for all branch operands. **WHILE** and **LEAVE** require an indefinite chain of addresses to resolve. This is done by chaining through the address space reserved by >**MARK**. If an implementation uses eight-bit offsets or other than sixteen-bit addresses, the implementation will change considerably. Where a **CODE** sequence is used, a sample high-level definition is also given. **CODE** sequences are used where execution speed is important in the implementation.

The above paragraphs discussing implementation got us ahead of ourselves by mentioning words that have not been defined. At this point, let's pick up some of these words and also discuss what happens in the Forth compiler when you want to compile any of the Forth control structures.

Low-Level Definitions

At the lowest level, four words in the System Extension Word Set Glossary control the placement of addresses in the compiled instruction stream of the program which Forth is compiling:

>**MARK** addr1 --- addr2

Used at the source of a forward branch. Typically used after a branching operation. >**MARK** compiles space in the dictionary for a branch address which will later be resolved by >**RESOLVE**, and extends a possible chain of forward references. >**MARK** is used by **IF**, **WHILE** and **LEAVE**.

<**MARK** --- addr1

Used at the destination of a backward branch. <**MARK** reserves space in the dictionary for a branch address which typically will be resolved later by <**RESOLVE**. It leaves the current address on the stack for use by <**RESOLVE**. <**MARK** is used by **BEGIN** and **DO**.

>**RESOLVE** addr1 ---

Used at the destination of forward branches. >**RESOLVE** resolves a possible chain of addresses starting at addr1, compiling in the current dictionary address, until no more addresses are left to resolve. >**RESOLVE** is used by **UNTIL**, **THEN** and **REPEAT**.

<**RESOLVE** addr1 addr2 ---

Used at the source of a backward branch.<**RESOLVE** compiles addr1 into the dictionary and then uses >**RESOLVE** to resolve any possible pending forward branches. <**RESOLVE** is used by **UNTIL** and **LOOP**.

The following words also belong in the System Extension Word Set Glossary. They control branching of the Forth program.

BRANCH ---

At run time, an unconditional branch operation is performed with the branch address data contained in the compilation stream immediately following the branch. Whether the data is an absolute address or an offset, or is one, two or more bytes long is implementation specific.

At compile time, this is typically generated with a **COMPILE BRANCH** followed by <**RESOLVE** or >**MARK**.

?BRANCH flag ---

At run time, a branch is performed if the contents of the stack are false; otherwise execution continues at the compilation address immediately following the branch address. See **BRANCH** for more details.

At compile time, Forth statements like **IF**, **WHILE** or **UNTIL** will perform a **COMPILE ?BRANCH** followed by <**RESOLVE** or >**MARK**.

The above words are all in the Forth-83 Standard. The following word is added to provide minimal orthogonality in the Forth computer. Tests for false and true are now provided and allow for more powerful high-level constructs.

TBRANCH flag ---

At run time, a branch is performed if the contents of the stack are true; otherwise execution continues immediately following the branch address. See **?BRANCH** for more details.

Additional branches could be considered.Such branches as <**BRANCH**, >**BRANCH**, =**BRANCH**, etc. are possible. However, since the Forth computer really only tests for true or false, and since the words <, > and = provide a suitable true or false flag, the words are not strictly necessary.

Now that we know how the compiler can compile in various branching instructions, let's see how high-level control structures are implemented.

All high-level control structures are contained in colon definitions. This means that the Forth compiler is in compile mode when it encounters a control structure statement. Since we generally do not want to compile the control statement but do want the control statement to compile appropriate code, these statements are made **IMMEDIATE** so they will execute even at compile time. The following example of **IF** will demonstrate the general technique.

```
: IF ( --- 0 addr1 2 )
0 0 ( mark chains )
COMPILE ?BRANCH ( compile ?branch )
>MARK 2 ; IMMEDIATE ( mark place )
```

IF first initializes two possible chains of addresses to zero or none. It then compiles **?BRANCH** so that a branch in the program will be taken if the top of the stack is zero or false. >**MARK** then marks the place in the instruction stream so that **THEN** (using >**RESOLVE**) can compile the proper forward address. Finally, two is left on the stack for structure checking by **ELSE** and **THEN**.Other control statements use the same general technique.

High-Level Definitions

The following words belong in the Required Word Set. The compiling words <**MARK**, >**MARK**, <**RESOLVE** and >**RESOLVE** are used as defined in the System Extension Word Set Glossary.

IF flag ---
--- sys (compiling)
Used in the form:
flag **IF ... (ELSE) ... THEN**

At execution time, if the flag is true, the words following **IF** are executed up to **ELSE** (if present). The words between **ELSE** and **THEN** are skipped. If the flag is false, words from **IF** through **ELSE** (or from **IF** through **THEN** if there is no **ELSE**) are skipped.

IF is an immediate compiling word. It compiles **?BRANCH** and uses >**MARK** to reserve space for a resolved forward branch. "sys" is system-dependent data to provide compiler security and address resolution information. **IF** must be balanced by a corresponding **ELSE** or **THEN**.

ELSE ---
sys1 --- sys2 (compiling)
Used in the form:
flag **IF ... ELSE ... THEN**

ELSE executes after the true part following **IF** if the flag is false and continues up to and through **THEN**.

ELSE is an immediate compiling word. It verifies, through sys1, that it follows an **IF**, compiles **BRANCH** and uses >**MARK** to reserve space for a resolved forward branch and uses >**RESOLVE** to fill in the reserved branch space following **IF**.

THEN ---
sys2 --- (compiling)
Used in the form:
IF ... ELSE ... THEN
or
IF ... THEN

THEN has no run-time action. It is the point where execution continues after **ELSE** (or after **IF** if there is no **ELSE**).

THEN is an immediate compiling word. It verifies that it correctly follows a corresponding **IF** or **ELSE** and then resolves the forward branches left by **IF** and/or **ELSE** with >**RESOLVE**.

BEGIN ---
--- sys (compiling)
Used in the form:
BEGIN ... (flag **WHILE**) ... (**LEAVE**) ... flag **UNTIL**
or
BEGIN ... (flag **WHILE**) ... (**LEAVE**) ... **REPEAT**

BEGIN marks the start of an uncounted loop that is terminated with either **UNTIL** for conditional looping or **REPEAT** for uncon-ditional looping. There is no run-time action.

BEGIN is an immediate compiling word. It marks the current compile address with <**MARK** for address resolution and leaves "sys" for proper compile-time action.

DO w1 w2 ---
--- sys (compiling)
Used in the form:
DO ... (flag **WHILE**) ... (**LEAVE**)... **LOOP**
or
DO ... (flag **WHILE**) ... (**LEAVE**)... **+LOOP**

DO marks the start of a counted loop. The loop begins at w2 and terminates at w1 based on the terminating conditions for **LOOP** or **+LOOP**. At run time, the loop parameters are removed from the parameter stack and used in a system-dependent manner to provide the proper loop control. A legal Forth-83 program cannot modify either the loop index or loop limit while within the loop.

DO is an immediate compiling word. It compiles a run-time do loop initializing word, marks the current compile address with <**MARK** and leaves "sys" for proper compile-time operation.

WHILE flag ---
sys1 --- sys2 (compiling)
Used in the form:
DO ... flag **WHILE ... (+)LOOP**
or
BEGIN ... flag **WHILE ... REPEAT**
or
BEGIN ... flag **WHILE ...** flag **UNTIL**

At run time, selects conditional execution based on "flag." If flag is true, execution continues after **WHILE**. If flag is false, execution continues after the **LOOP**, **+LOOP**, **REPEAT** or **UNTIL** terminating the **DO** or **BEGIN**, thus exiting the loop structure. It is an implementation consideration whether **WHILE** removes loop parameters when it exits a counted loop.

WHILE is an immediate compiling word. It compiles **?BRANCH** and uses >**MARK** to reserve space for a resolved forward branch. "sys" is system-dependent data to provide compiler security and address resolution information.

LEAVE flag ---
sys1 --- sys2 (compiling)
Used in the form:
DO ... LEAVE ... (+)LOOP
or
BEGIN ... LEAVE ... REPEAT
or
BEGIN ... LEAVE ... flag **UNTIL**

At run time, execution continues after the **LOOP**, **+LOOP**, **REPEAT** or **UNTIL** terminating the **DO** or **BEGIN** at this syntactic nesting level, thus exiting the loop structure. It is an implementation consideration whether **LEAVE** removes loop parameters when it exits a counted loop.

LEAVE is an immediate compiling word. It compiles **BRANCH** and uses >**MARK** to reserve space for a resolved forward branch. "sys" is system-dependent data to provide compiler security and address resolution information.

REPEAT ---
sys --- (compiling)
Used in the form:
BEGIN ... (flag **WHILE**) ... (**LEAVE**) ... **REPEAT**

Marks the end of a **BEGIN REPEAT** loop. At execution time, **REPEAT** continues execution to just after the corresponding **BEGIN**.

REPEAT is an immediate compiling word.It compiles **BRANCH** and then uses <**RESOLVE** to resolve the backward branch to the corresponding **BEGIN** and >**RESOLVE** to resolve any possible **WHILE**s or **LEAVE**s. "sys" is used to ensure proper **BEGIN REPEAT** nesting.

UNTIL flag ---
sys --- (compiling)
Used in the form:
BEGIN ... (flag **WHILE**) ... (**LEAVE**) ... flag **UNTIL**

Marks the end of a **BEGIN UNTIL** loop. At execution time, **UNTIL** continues execution to just after the corresponding **BEGIN** if flag is false.If flag is true, looping is terminated.

UNTIL is an immediate compiling word. It compiles **?BRANCH** and then uses <**RESOLVE** to resolve the backward branch to the corresponding **BEGIN** and >**RESOLVE** to resolve any possible **WHILE**s or **LEAVE**s. "sys" is used to ensure proper **BEGIN UNTIL** nesting.

LOOP ---
sys --- (compiling)

At execution time, increments the **DO LOOP** index by one. If the new index was incremented across the boundary between limit-1 and limit, execution continues at the compilation address immediately following the branch address. If the loop is not terminated, execution continues to just after the corresponding **DO**. The loop arguments must be removed in a system-dependent manner when exiting the loop structure.

LOOP is an immediate compiling word. It compiles a run-time loop word and then uses <**RESOLVE** to resolve the backward branch to the corresponding **DO** and >**RESOLVE** to resolve any possible **WHILE**s or **LEAVE**s. "sys" is used to ensure proper **DO LOOP** nesting.

+LOOP n ---
sys --- (compiling)

At execution time, increments the **DO LOOP** index by n. If the new index was incremented across the boundary between limit-1 and limit, execution continues at the compilation address immediately following the branch address. If the loop is not terminated, execution continues to just after the corresponding **DO**. The loop arguments must be removed in a system-dependent manner when exiting the loop structure.

+LOOP is an immediate compiling word. It compiles a run-time **+LOOP** word and then uses <**RESOLVE** to resolve the backward branch to the corresponding **DO** and >**RESOLVE** to resolve any possible **WHILE**s or **LEAVE**s. "sys" is used to ensure proper **DO +LOOP** nesting.

The eagle-eyed among you have noticed that the wording of the definitions for the control structures allows a more general use of the words in Forth programs than specified by the Forth-83 Standard. Following are the Forth-83 allowed structures and the generalized structures. The generalized structures contain the Forth-83 structures as a subset, but are considerably more powerful and regular.

Forth-83 definitions:

IF... (ELSE)... THEN
BEGIN ... (WHILE) ... REPEAT
BEGIN ...UNTIL
DO... (LEAVE) ... (+)LOOP

Generalized definitions:

IF... (ELSE)... THEN
BEGIN ... (WHILE/LEAVE) ... REPEAT/UNTIL
DO... (WHILE/LEAVE) ... (+)LOOP

There are quite a few advantages to having the more regular control structures. In the real world, the standard Forth control structures force awkward programming with **OR**, **AND** and **0=** to concatenate and invert the intermediate testing points to get to a single decision point. Not only does this lead to obscure programming, but it complicates maintenance.

Also, the old Forth **BEGIN AGAIN** infinite loop structure (which is removed to the uncontrolled reference word set even though the Forth nucleus used it) can now be expressed with the natural **BEGIN REPEAT** loop structure. Also, all loops can be exited with **WHILE/LEAVE** (instead of you having to remember which type of loop you are in).

Now, remember that we defined a low-level machine word **TBRANCH** that branched on the stack being true rather than false. This orthogonalized the basic Forth machine instructions but, so far, no high-level Forth control words have used this low-level word. We will now proceed to orthogonalize the high-level Forth control structures.

Forth programmers can look through their code (or through the Forth nucleus) and see many instances of **0= IF**, **0= WHILE** or **0= UNTIL**. These code sequences are necessary to manipulate the stack value so that the limited branching capability of the Forth computer can be used. (The other problem, having to **OR** or **AND** together multiple decisions before a **WHILE** or **UNTIL** has been lifted already with the revised definitions.) What the Forth language is really implementing is **IFTRUE** or **WHILETRUE** or **UNTILFALSE** and we would like the corresponding definitions **IFFALSE** or **WHILEFALSE** or **UNTILTRUE**. However, because of the history of usage of the Forth words, we will define new words **NIF** (not if or if false), **NWHILE** (not while or while false) and **NUNTIL** (not until or until true).

NIF flag ---
--- sys (compiling)
Used in the form:
flag **NIF ... (ELSE) ... THEN**

At execution time, if flag is false, the words following **NIF** are executed up to **ELSE** (if present). The words between **ELSE** and **THEN** are skipped. If flag is true, words from **NIF** through **ELSE** or **NIF** through **THEN** are skipped.

NIF is an immediate compiling word. It compiles **TBRANCH** and uses >**MARK** to reserve space for a resolved forward branch. "sys" is system-dependent data to provide compiler security and address resolution information. **NIF** must be balanced by a corresponding **ELSE** or **THEN**.

NWHILE flag ---
sys1 --- sys2 (compiling)
Used in the form:
DO ... flag **NWHILE** ... **(+)LOOP**
or
BEGIN ... flag **NWHILE** ... **REPEAT**
or
BEGIN ... flag **NWHILE** ... flag **UNTIL**

At run time, selects conditional execution based on "flag." If flag is false, execution continues after **NWHILE**. If flag is true, execution continues after the **LOOP**, **+LOOP**, **REPEAT** or **UNTIL** terminating this **DO** or **BEGIN**, thus exiting the loop structure. It is an implementation consideration whether **NWHILE** removes loop parameters when it exits a counted loop.

NWHILE is an immediate compiling word. It compiles **TBRANCH** and uses >**MARK** to reserve space for a resolved forward branch. "sys" is system-dependent data to provide compiler security and address resolution information.

NUNTIL flag ---
sys --- (compiling)
Used in the form:
BEGIN ... (flag **(N)WHILE**)...
(**LEAVE**) ... flag **NUNTIL**

Marks the end of a **BEGIN NUNTIL** loop. At execution time, **NUNTIL** continues execution to just after the corresponding **BEGIN** if flag is true. If flag is false, looping is terminated.

NUNTIL is an immediate compiling word. It compiles **TBRANCH** and then uses <**RESOLVE** to resolve the backward branch to the corresponding **BEGIN** and >**RESOLVE** to resolve any possible **(N)WHILE**s or **LEAVE**s. "sys" is used to ensure proper **BEGIN NUNTIL** nesting.

So far, we have regularized the looping control structures so that awkward sequences of >**R**, **R**>, **OR**, **AND** and **0=** are not necessary before **WHILE**s or **UNTIL**s to concatenate multiple decisions into one decision point. The same needs to be done for **IF ELSE THEN** since it is a pain to either concatenate decisions before one **IF** statement or to nest **IF**s with the task of keeping track of the **IF**s and using the appropriate number of **THEN**s at the end. Thus, we will define **THENIF** (to and decisions) and **NTHENIF** (to or decisions). Used properly, these constructs allow rather simple expressions of logic that were either very difficult or hard to understand in regular Forth, to be expressed in a readable manner.

THENIF flag ---
sys1 --- sys2 (compiling)
Used in the form:
flag **IF** ... (**ELSE**) ... flag **THENIF** ...
(**ELSE**) ... **THEN**

If flag is true, the words following **THENIF** are executed up to the **ELSE** (if present). The words between **ELSE** and **THEN** are skipped. If flag is false, words from **THENIF** through **ELSE** or **THEN** are skipped. **THEN** is the point where execution continues after **ELSE** (or after **THENIF** if there is no **ELSE**).

THENIF is an immediate compiling word.It verifies that it correctly follows a corresponding **IF**, **ELSE** or **THENIF**, compiles **?BRANCH** and uses >**MARK** to reserve space for a resolved forward branch.

NTHENIF flag ---
sys1 --- sys2 (compiling)
Used in the form:
IF ... (**ELSE**) ... **NTHENIF** ... (**ELSE**) ...
THEN

If flag is false, the words following **NTHENIF** are executed up to the **ELSE** (if present). The words between **ELSE** and **THEN** are skipped. If flag is true, words from **NTHENIF** through **ELSE** or **THEN** are skipped. **THEN** is the point where execution continues after **ELSE** or after **(N)IF** or **(N)THENIF** if there is no **ELSE**.

NTHENIF is an immediate compiling word. It verifies that it correctly follows a corresponding **IF**, **ELSE** or **(N)THENIF**, compiles **TBRANCH** and uses >**MARK** to reserve space for a resolved forward branch.

THENIF considerably lightens the load of writing nested **IF** structures. Another form of nested **IF**s is the **CASE** structure. While a **CASE** can be simulated with **THENIF**s, the following word definitions make for more easily understandable code. **CASE**s have been covered in several previous Forth publications, so that the following code illustrates how your particular **CASE** can be integrated. Also, the words chosen can be changed to meet your requirements (i.e. **CASE ESAC**, **OF ENDOF** or { } for the case delimiters).

CASES n --- n
--- sys (compiling)
Used in the form:
n ... **CASES**
n1 ... **CASE** ... **ENDCASE**
n2 ... **CASE** ... **ENDCASE**
...
nx ... **CASE** ... **ENDCASE**
ENDCASES

CASES has no execution-time effect.

CASES is an immediate compiling word. It provides proper system information for the following **CASE ENDCASE ENDCASES** words.

CASE n n1 --- n |-
sys1 --- sys2 (compiling)
Used in the form:
n1 **CASE** ... **ENDCASE**

At execution time, **CASE** examines the top two stack items. If they are equal, they are discarded and execution continues immediately following the branch address that follows the run-time **CASE** word. If the two top stack items are not equal, the top item is discarded and a branch is performed to the location following the **ENDCASE** word.

CASE is an immediate compiling word. It verifies that it followed a **CASES** or **ENDCASE** word, compiles the run-time **CASE** word and uses >**MARK** to reserve space for a resolved forward branch.

ENDCASE ---
sys1 --- sys2 (compiling)
Used in the form:
n1 **CASE** ... **ENDCASE**

At execution time, **ENDCASE** performs a branch to the end of the **CASE** structure. See **BRANCH**.

ENDCASE is an immediate compiling word. It verifies that it followed a **CASE** word and uses **ELSE** to compile **BRANCH**, >**MARK** to mark the forward reference and >**RESOLVE** to resolve the branch from the previous **CASE** to the current compilation address.

ENDCASES n ---
sys1 --- (compiling)
Used in the form:
n ... **CASES**
n1 ... **CASE** ... **ENDCASE**
n2 ... **CASE** ... **ENDCASE**
...
nx ... **CASE** ... **ENDCASE**
ENDCASES

At execution time, **ENDCASES** pops the case selector value from the stack since no case selection criterion has been met.

ENDCASES is an immediate compiling word. It verifies that it followed a **CASES** or **ENDCASE**, compiles **DROP** and then uses **THEN** logic to resolve all the **ENDCASE** branches to the current compilation address.

Summary

We have seen how the Forth computer implements changes in program flow and how the Forth language uses this capability to implement high-level language control constructs. The constructs implemented are:

(N)IF ... (ELSE) ... ((N)THENIF) ... (ELSE) ... THEN
BEGIN ... ((N)WHILE/LEAVE) ... (N) UNTIL/REPEAT
DO... ((N)WHILE/LEAVE) ... (+)LOOP

The Forth computer has been orthogonalized by giving it the ability to branch on a true condition as well as on a false condition. The Forth computer can now:

1. **BRANCH** always.
2. **TBRANCH** if the stack is true.
3. **?BRANCH** if the stack is false (**FBRANCH**).

Simple Implementation of Forth Control Structures

```
: >MARK                          ( addr1 --- addr2                 )
  ?COMP HERE SWAP , ;            ( extend chain of forward refs    )

: <MARK                          ( --- addr1 0                     )
  ?COMP HERE 0 ;                 ( where to branch, init chain     )

: >RESOLVE                       ( addr1 ---                       )
  BEGIN
  ?DUP WHILE                     ( chain not resolved yet?         )
  DUP @                          ( get where it points to          )
  HERE ROT ! REPEAT ;            ( resolve it to HERE              )

: <RESOLVE                       ( addr1 addr2 ---                 )
  SWAP ,                         ( put backward addr in            )
  >RESOLVE ;                     ( and resolve any forward refs    )

CODE BRANCH                      (    ---                          )
     B H MOV  C L MOV            ( move IP                         )
     M C MOV  H INX  M B MOV     ( get next address                )
     NEXT ;C                     ( NEXT is a macro, >NEXT JMP      )

: BRANCH                         ( branch to addr following        )
  R> @ >R ;                      ( get addr at branch              )

CODE ?BRANCH                     ( flag ---                        )
     H POP  L A MOV  H ORA       ( is flag false (0)?              )
     ' BRANCH >BODY JZ           ( yes, do branch logic            )
     B INX  B INX                ( no, skip over branch addr       )
     NEXT ;C

: ?BRANCH                        ( branch if false                 )
  R> SWAP
  NIF @                          ( stack false, get addr           )
  ELSE 2+                        ( true, skip                      )
  THEN  >R ;

CODE TBRANCH                     ( flag ---                        )
     H POP  L A MOV  H ORA       ( is flag true (#0)?              )
     ' BRANCH >BODY JNZ          ( yes, do branch logic            )
     B INX  B INX                ( no, skip over branch addr       )
     NEXT ;C

: TBRANCH                        ( branch if true                  )
  R> SWAP
  IF @
  ELSE 2+
  THEN >R ;

: (DO)                           ( w1 w2 ---                       )
  R> -ROT                        ( put return addr below loop parm)
  SWAP DUP >R                    ( limit value                     )
  - >R                           ( negative # iterations           )
  >R ;                           ( original return stack           )

CODE (LOOP)                      ( ---                             )
     RPP LHLD  M INR             ( inc # left                      )
     ' BRANCH >BODY JNZ          ( lsb not 0 go to branch          )
     H INX  M INR                ( inc msb                         )
     ' BRANCH >BODY JNZ          ( msb not zero branch             )
LABEL loop-term
     B INX B INX B INX B INX     ( else skip branch addr & 1-ex    )
LABEL loop-exit
     RPP LHLD  4 D LXI  D DAD    ( inc return stack past loop parm)
     RPP SHLD  NEXT  ;C          ( and store                       )
```

```
: (LOOP)
  R> R> 1+                  ( inc # times left              )
  DUP >R                    ( replace                       )
  IF @                      ( not end branch                )
  ELSE 2+                   ( end, skip over branch addr    )
  THEN >R ;                 ( replace and return            )

CODE LOOP-EXIT              (  ---                          )
     loop-exit HERE 2- ! ;C ( use loop-exit for LOOP-EXIT   )

: LOOP-EXIT
  R> R> R>                  ( get loop parms                )
  2DROP >R ;                ( toss them                     )

CODE (+LOOP)                ( n ---                         )
     RPP LHLD  M E MOV  H INX  M D MOV  ( get # left        )
     H POP  H A MOV  A ORA  ( find out if pos or neg        )
     0< NOT IF  D DAD       ( pos add in                    )
     loop-term JC           ( and term if cross 0           )
LABEL loop-branch
     XCHG  RPP LHLD  E M MOV  H INX  D M MOV ( replace # left )
     ' BRANCH >BODY JMP THEN ( and branch to do             )
     D DAD  loop-branch JC  ( neg inc, not cross go         )
     loop-term JMP ;C       ( crossed, term loop            )

: (+LOOP)                   ( n ---                         )
  R> SWAP R> 2DUP +         ( get # left and dir            )
  >R OVER XOR 0<            ( dir, # left different?        )
  IF R@ XOR 0<              ( yes, dir, # left diff sign    )
  IF @                      ( yes, not end, loop            )
  ELSE 2+                   ( no, end of loop               )
  THEN
  ELSE DROP @               ( dir, # left same sign, loop   )
  THEN >R ;                 (                               )

CODE I                      ( --- n                         )
     RPP LHLD  M E MOV  H INX  M D MOV  ( # left            )
     H INX  M A MOV  H INX  M H MOV     ( limit             )
     A L MOV  D DAD  HPUSH JMP  ;C      ( add to get index  )

: I
  R> R> R@                  ( get loop parms                )
  +                         ( add to get index              )
  -ROT R> R> ;              ( put stuff back, leaving index )

CODE J
     RPP LHLD  4 D LXI  D DAD  ( point to inner loop parms  )
     ' I 5 + JMP ;C            ( and to I logic             )

: J
  R> R> R> R> R@ + -ROT >R >R -ROT >R >R ;

: BEGIN                     ( --- addr1 0 1                 )
  <MARK 1  ; IMMEDIATE      ( mark start and BEGIN loop     )

: IF                        ( --- 0 addr1 2                 )
  0 0                       ( mark chains                   )
  COMPILE ?BRANCH           ( compile ?branch               )
  >MARK 2  ; IMMEDIATE      ( mark place                    )

: NIF                       ( --- 0 addr1 2                 )
  0 0                       ( mark chains                   )
  COMPILE TBRANCH           ( compile tbranch               )
  >MARK 2  ; IMMEDIATE      ( mark place                    )

: DO                        ( --- addr1 0 3                 )
  COMPILE (DO)              ( run time do                   )
  <MARK 3  ; IMMEDIATE      ( mark place                    )

: ELSE                      ( 0 addr1 2 --- addr2 0 -2      )
  2 ?PAIRS                  ( follow if?                    )
  COMPILE BRANCH            ( branch                        )
  SWAP >MARK SWAP           ( extend chain of elses         )
  >RESOLVE                  ( resolve if                    )
  0 -2  ; IMMEDIATE         ( init chain and else flag      )
```

The Forth-83 language control structures have been generalized, made easier to use, more powerful, modeless and orthogonal while retaining Forth-83 compatibility as follows:

- **NIF**, **THENIF** and **NTHENIF** have been added to **IF ELSE THEN**. This allows direct testing for both true and false stack values as well as easy nesting of **IF** structures. A sample non-indexed **CASE** structure has been added.

- Uncounted loops always start with **BEGIN**. They can terminate with the same three conditions that the Forth computer allows:
  REPEAT loops back always.
  UNTIL loops back if the stack is false.
  NUNTIL loops back if the stack is true.

- Both counted and uncounted loops can be exited with the same set of words, can have multiple exits and can exit with the same three conditions that the Forth computer allows:

  LEAVE exits always.
  WHILE exits if the stack is false.
  NWHILE exits if the stack is true.

Bibliography

Forth-83 Standard

"A Generalized Loop Construct for Forth," *Forth Dimensions*, Vol. II, No.2.

"Case Contest," *Forth Dimensions*, Vol.II, No. 3.

"Transportable Control Structures," 1981 Rochester Forth Standards Conference.

"Modern Control Logic," Wil Baden.

C64-FORTH/79 New and Improved for the Commodore 64

C64-FORTH/79™ for the Commodore 64-
$99.95

- •New and improved FORTH-79 implementation with extensions.
- •Extension package including lines, circles, scaling, windowing, mixed high res-character graphics and sprite graphics.
- •Fully compatible floating point package including arithmetic, relational, logical and transcendental functions.
- •String extensions including LEFT$, RIGHT$, and MID$.
- •Full feature screen editor and macro assembler.
- •Compatible with VIC peripherals including disks, data set, modem, printer and cartridge.
- •Expanded 167 page manual with examples and application screens.
- •"SAVE TURNKEY" normally allows application program distribution without licensing or royalties.

(Commodore 64 is a trademark of Commodore)

TO ORDER

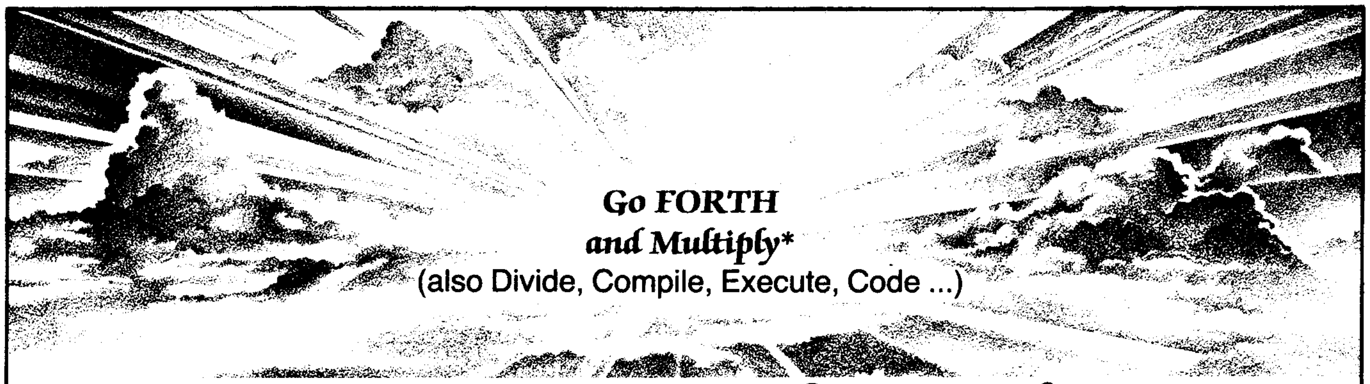
-Disk only.
-Check, money order, bank card, COD's add $1.65
-Add $4.00 postage and handling in USA and Canada
-Mass. orders add 5% sales tax
-Foreign orders add 20% shipping and handling
-Dealer inquiries welcome

PERFORMANCE MICRO PRODUCTS

770 Dedham Street,
Canton, MA 02021
(617) 828-1209

```
: THEN                                addr1 addr2 2|-2 ---               )
  ABS 2 ?PAIRS                        follow if or else?                 )
  >RESOLVE                            resolve if                         )
  >RESOLVE ; IMMEDIATE                resolve else                       )

: THENIF                              a1 a2 2|-2 --- a1 a3 2             )
  ABS 2 ?PAIRS                        follow if or else?                 )
  COMPILE ?BRANCH                     compile ?branch                    )
  >MARK 2  ; IMMEDIATE                mark place, extend chain           )

: NTHENIF                             a1 a2 2|-2 --- a1 a3 2             )
  ABS 2 ?PAIRS                        follow if or else?                 )
  COMPILE TBRANCH                     compile tbranch                    )
  >MARK 2  ; IMMEDIATE                mark place, extend chain           )

: cleave                              addr1 1|3 addr2 --- addr3 1|3      )
                                      compile branch type passed         )
  -1 >R                               mark start of loop                 )
  BEGIN DUP >R                        put parm on return                 )
  3 = NWHILE                          if do flag exit loop               )
  R@ 1- NUNTIL                        else check for until               )
  >MARK BEGIN                         extend chain                       )
  R> DUP 1+                           return stack to parm stack         )
  NUNTIL DROP ;                       until -1 encountered               )

: LEAVE                               addr1 1|3 --- addr2 1|3            )
  [']  BRANCH                         get branch                         )
  cleave  ; IMMEDIATE                 compile it and extend chain        )

: WHILE                               addr1 1|3 --- add2 1|3             )
  ['] ?BRANCH                         get ?branch                        )
  cleave  ; IMMEDIATE                 compile it and extend chain        )

: NWHILE                              addr1 1|3 --- add2 1|3             )
  ['] TBRANCH                         get tbranch                        )
  cleave  ; IMMEDIATE                 compile it and extend chain        )

  UNTIL                               addr1 addr2 1 ---                  )
  1 ?PAIRS                            follow begin?                      )
  COMPILE ?BRANCH                     compile ?branch                    )
  <RESOLVE    ; IMMEDIATE             resolve to begin,while/leave's     )

: NUNTIL                              addr1 addr2 1 ---                  )
  1 ?PAIRS                            follow begin?                      )
  COMPILE TBRANCH                     compile tbranch                    )
  <RESOLVE    ; IMMEDIATE             resolve to begin,while/leave's     )

: REPEAT                              addr1 addr2 1 ---
  1 ?PAIRS                            follow begin?                      )
  COMPILE BRANCH                      compile branch                     )
  <RESOLVE    ; IMMEDIATE             resolve to begin,while/leave's     )

: LOOP                                addr1 addr2 3 ---                  )
  3 ?PAIRS                            follow do?                         )
  COMPILE (LOOP)                      run time loop                      )
  <RESOLVE                            resolve to do and while/leave's)
  COMPILE LOOP-EXIT ;                 dump loop parms                    )
  IMMEDIATE

: +LOOP                               addr1 addr2 3 ---                  )
  3 ?PAIRS                            follow do?                         )
  COMPILE (+LOOP)                     run time +loop                     )
  <RESOLVE                            resolve to do and while/leave's)
  COMPILE LOOP-EXIT ;                 dump loop parms                    )
  IMMEDIATE

CODE (CS)                             n n1 --- n | -                     )
     H POP  D POP                     get args                           )
     L A MOV  E SUB  A L MOV  ( subtract them                            )
     H A MOV  D SBB  L ORA 0= NOT ( =?                                   )
     IF D PUSH  ' BRANCH >BODY JMP ( no, push n, br to nxt test)
     THEN  B INX B INX          ( yes, skip branch addr                  )
     NEXT  ;C
```

```
: (CS)                           ( n n1 --- n | -              )
  OVER =                         ( n=n1?                       )
  IF DROP R> 2+                  ( yes, drop n, skip branch addr )
  ELSE R> @                      ( no, branch                  )
  THEN >R ;

: CASES                          ( --- 0 0 4                   )
  0 0 4 ; IMMEDIATE              ( mark addr chains, cases     )

: CASE                           ( addr1 addr2 4 --- addr1 addr3 5)
  4 ?PAIRS                       ( follow cases or endcase?    )
  COMPILE (CS)                   ( run time case               )
  >MARK 5 ; IMMEDIATE            ( extend chain, mark case     )

: ENDCASE                        ( addr1 addr2 5 --- addr3 0 4   )
  5 ?PAIRS                       ( follow case?                )
  2 [COMPILE] ELSE               ( use else to compile branch,mark)
  DROP 4 ; IMMEDIATE             ( drop 2, mark endcase        )

: ENDCASES                       ( addr1 0 4 ---               )
  4 ?PAIRS                       ( follow cases or endcase     )
  COMPILE DROP                   ( drop n at run time          )
  >RESOLVE >RESOLVE ; IMMEDIATE ( resolve endcase's            )
```



Have You Gotten The Word Yet?

Companies such as IBM, Atari, Varian, Hewlett Packard, Dysan and Memorex are now using FORTH for a number of applications. If you are concerned about efficiency and transportability, then FORTH is a language you should learn.

Join the FORTH Revolution!

- Intensive 5-day workshops
- Small classes
- Experienced professionals
- On-site classes by special arrangement

FORTH Fundamentals	**$395.00**
Advanced Systems & Tools	**$495.00**

(For further information, please send for our complete FORTH workshop catalogue).

Inner Access Corporation
P.O. Box 888, Belmont, CA 94002
(415) 591-8295

Forth in Rehabilitation Applications

David L. Jaffe
Palo Alto, California

Readers of this magazine are well aware of the many computer applications of Forth. In this article, I will discuss the use of Forth in devices that serve disabled individuals, with a specific example drawn from my research.

First of all, a few definitions are in order. For the purpose of this paper, disabled people are those mentally normal individuals who experience physical difficulty in interacting with others or with their environment. This diminished interaction may include such output functions as mobility and communications (both person-to-person and person-to-machine) and input sensory functions of hearing and sight.

Devices for disabled people typically foster interaction between the user and the environment and other people by providing enhanced or alternate communication pathways. For example, some blind people use canes as a tactile hand extension to feel objects in their travel route. In addition, the echo of cane tap sounds can provide aural information about the surroundings. The utilization of the cane is thus a substitute for vision, while hearing aids or glasses provide enhanced hearing or vision by amplification of sound too soft for some to hear or correction of light rays that would be out of focus for misshaped eyeballs.

Adaptations of commercially available equipment built for the able bodied can often assist disabled people. An electric typewriter operated by a mouthstick is a writing substitute for people who do not have use of their hands or arms. Even toys can be useful; a Speak-and-Spell can produce spelled speech for non-vocal individuals. In these examples a machine facilitates communication between individuals. Other disabilities require specialized solutions. An electric wheelchair provides mobility for those who are unable to command their legs. In this example, a hand-operated joystick is typically used to tap a control site, the user's hand position, and translate it into mechanical rotation of electric motors, producing motion.

The formula of a user interface to a machine, translation of sensory input into machine activation, and production of a result which reflects on the environment is common to many rehabilitation devices. (Figure one). These elements are identical to those constituting a process control system. Thus, one strategy for developing aids for disabled individuals is to use methodologies that have been successful in process control applications; an area in which Forth and microcomputer technology excels.

Although many current products for the disabled do not use a computer, there is a class of rehabilitation devices that could benefit from the inclusion of a microcomputer. Products that require 1) an unusual user input, 2) a complex translation algorithm, 3) a real-time output, or 4) that are dynamically programmable are good candidates for an embedded microcomputer. For rehabilitation devices developed with this scheme, the user interacts with it through specialized inputs and outputs rather than the keyboards, cassette recorders, disk drives and CRT terminals of consumer microcomputer systems. The intricacies of the computer's operation need not be of concern to the user. In fact, from the users' point of view, it may even be desirable to be unaware of the existence of the embedded computer; it functions as a "black box." An example of such a system would be a voice-activated robotic aid in which vocal commands of the user are translated by an internal computer into mechanical movement instructions.

The design and development of any microcomputer product is not an easy task. Devices for rehabilitation use are no exception. As with traditional systems, the rehabilitation design engineer must be concerned with both hardware and software issues. Besides resource, performance and cost considerations (memory requirements, program execution time and product price), an additional consideration involves the design of microcomputer hardware and software that is accessible to the researcher during development but performs invisibly for the end user.

At the Palo Alto Veterans Administration Medical Center's Rehabilitation Research and Development center, standardized hardware, software and an efficient development strategy has been formulated for the construction of embedded microcomputer systems. Key elements of this process are listed below.

1. Industry standard computer boards are selected wherever possible. Although this may be more costly in price and space than building a custom board, it

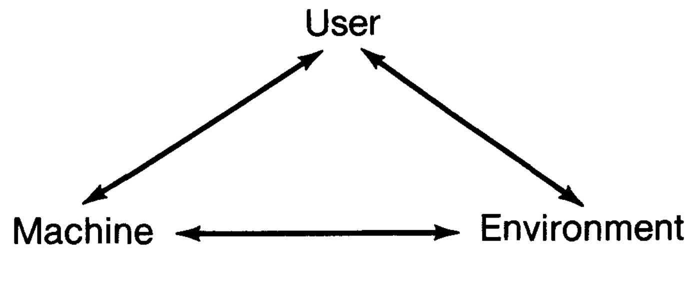


Figure One
Interaction Triad

does assure the designer of a working piece of hardware. It also reduces the designing, wiring, testing and debugging time necessary in building a custom board. Standard bus boards are commercially available and can provide almost any function: serial interface, parallel interface, memory, counter/timer, A/D or D/A conversion. Their widespread availability means that systems specifying these boards can be easily replicated.

2. As previously mentioned, Forth has proven to be a software system that is ideal for process control applications. A major advantage of the language for rehabilitation purposes is that its interactive development techniques and bottom-up programming structure can foster rapid implementation of systems that can be demonstrated to prospective users (and to superiors) during their development phase. During this time, program segments can be easily modified and optimized. The resultant Forth application program typically runs faster than the comparable BASIC program and is easier to code than the assembly language equivalent.

3. The final element in this design methodology is the link between a computer development system and the rehabilitation application system. This is done in the following manner. Hardware communication is typically accomplished by a serial interface between the systems. During development, this connection is the pathway for Forth dialog, while during operation it serves as a serial maintenance link providing a means for parameter modification and trouble-shooting. Figure two diagrams the relationship of the host and target systems and their memory resources during both the development and rehabilitation use phases.

Forth uses this link in several ways. The host development system with its disk, printer and terminal resources is used to edit, store and list the programmer's code. When entered, the program is transmitted over the link and compiled by the application's Forth kernel into its memory space. At this point the program can be tested on the application system, using the hardware that will comprise the final system. When thoroughly tested, the application program RAM memory image is transferred into EPROM storage. Then, upon powerup, the target system begins executing the application program, now in non-volatile memory. The version of Forth sold by Jib Ray supports the Host/Target scheme presented here.[1]

The typical minimum hardware requirements for the host development system are a CP/M microcomputer system with disk storage, CRT terminal and a serial communications link to the application system. The target system also requires a serial port, as well as EPROM memory to hold the Forth kernel, and memory that can be configured either as random access or nonvolatile storage. Several commercially standard products are available that contain this functionality on a single board. Additional boards to support the desired application would complete the target system's complement of hardware.

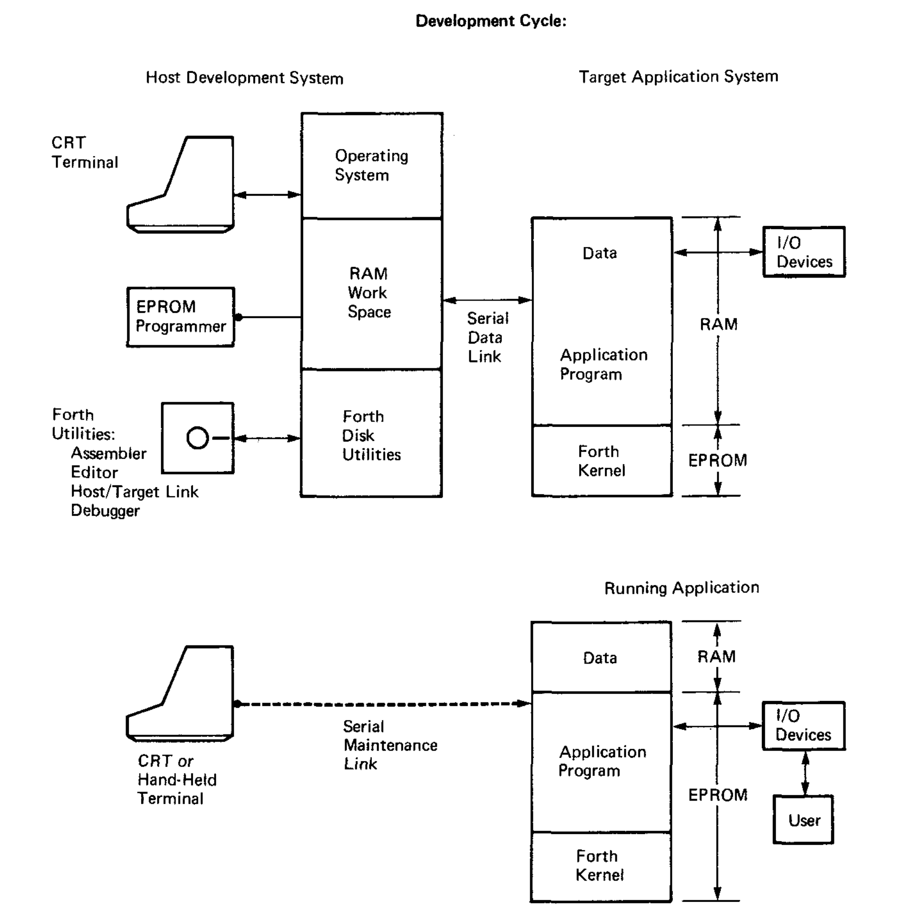


Figure Two
Host/Target System Development

Specific Example
Ultrasonic Head Control Unit

The Palo Alto VA Rehabilitation Research and Development Center recognized and addressed the generic need of disabled people to communicate their will to their environment.

In the specific design that follows, a unique man/machine has been developed using standard microcomputer hardware and Forth software techniques. The result is a unit with which severely disabled users can control devices. The unit translates head position information into control signals which operate devices to which it is attached, such as electric wheelchairs or specialized communications systems.

In this design, two Polaroid ultrasonic transducers are employed (figure three). They emit inaudible sound waves which propogate through the air until reflected by an object. A portion of the echo signal returns to the transmitting sensor and is detected by an electronic circuit. The measured time from transmission of the ultrasound pulse to the reception of the echo is proportional to the round-trip distance from the sensor to the object. In the commercial camera application, camera focussing is accomplished by ranging the distance from the camera to the subject being photographed. In this rehabilitation application, two separated sensors are directed at the user's head (from the front or the rear). The two distance ranges, one from each sensor to the head, and the fixed separation of the sensors describe an imaginary triangle whose vertices are the two stationary sensors and the user's head. A geometric relationship allows the offset from the base line and center line of the two sensors to be calculated. This information is then used to map the user's head position into a two-dimensional control space.

In operation, users of the Ultrasonic Head Control Unit (UHCU) merely tilt their head off the vertical axis in the forward/backward or left/right directions. Their changing head position can be made to produce output signals identical to those from a proportional joystick. Both these interfaces, the UHCU and the joystick, can be used to control devices to which they are attached such as an electric wheelchair, a communication aid or a video game.

An examination of the last dictionary word (figure four) will help to explain the unit's operation in a mobility application on an electric wheelchair.

The Forth software operates in one continuous loop which is entered upon power-up. At that time, the system is intialized; all the hardware ports are configured and variables are set to their starting values. A short audible tone signals the unit is ready for operation.

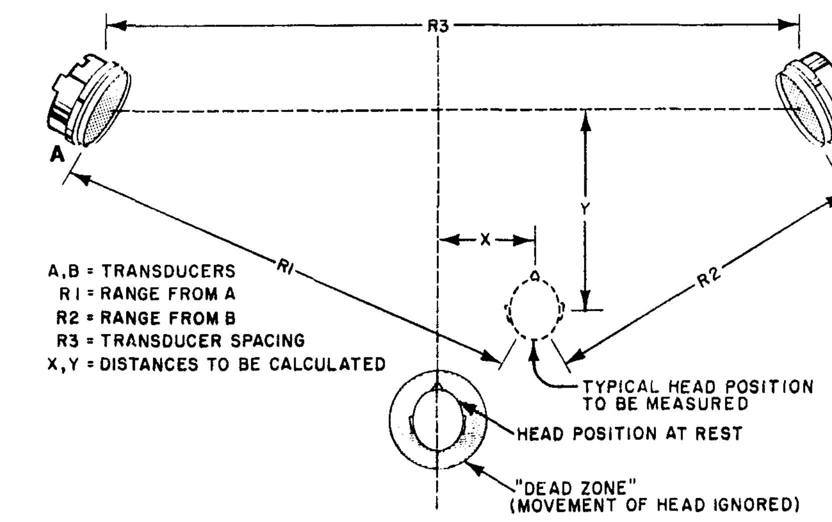

R3
A
B
Y
X
R1
R2
A,B = TRANSDUCERS
R1 = RANGE FROM A
R2 = RANGE FROM B
R3 = TRANSDUCER SPACING
X,Y = DISTANCES TO BE CALCULATED
TYPICAL HEAD POSITION TO BE MEASURED
HEAD POSITION AT REST
"DEAD ZONE" (MOVEMENT OF HEAD IGNORED)

Figure Three
Ultrasonic Head Control
Unit geometry

?SWITCH looks for a change of state of a head-operated switch. When it has been pressed, the buzzer provides an acknowledgement and the Polaroid ranging electronics are activated. After a brief delay, the user's head position is acquired and used as a reference from which to judge subsequent head positions. After this **CENTER**ing operation, the motor controller is powered by the software activation of a relay. At this point, the operating loop of the software is entered. First a pair of distance rangings to the head are made and averaged with previous rangings. This procedure assures a smooth ride despite an occasional bad data value or rapidly changing head position due to a bump in the path of the wheelchair or a sneezing rider. The **CALC** routine figures the absolute X-Y head position, while **ADJUST** subtracts the current head position from the reference head positions, thus producing relative data. **VELOCITY** and **SPEED** calculate the percentage of the maximum wheelchair velocity desired and output the appropriate values to the digital-to-analog converters that produce such velocity through the action of the wheelchair motor controller. At the end of the loop, the head switch is again interrogated for a state change, which would signify that the user wished to stop the wheelchair. If so, the motor controller and ranging system are deactivated.

The main advantage of this type of hardware interface is no mechanical contact between the sensors and the user's head is required. This effectively separates the user from the device being controlled. Therefore with this unit, users should not feel "wired-up" or confined by a device around their face or body, as frequently occurs with other interfaces. The UHCU as implemented on an electric wheelchair has another advantage; it is aesthetically superior to competing man/machine interfaces used for this purpose. It has also proven to be more socially acceptable than other alternatives.

Another desirable characteristic of the UHCU is its high speed of operation (which, in part, can be attributed to its use of Forth). Devices that it controls can thus be manipulated quickly. In addition, the UHCU can be directly substituted in many applications where a joystick is currently used. Its real-time

```
:  RUN
     INITIALIZE    BUZZ    2 SEC-DELAY
     BEGIN

     BEGIN    ?SWITCH   UNTIL

     BUZZ    2 SEC-DELAY   RANGING-ON   CENTER BUZZ CHAIR-ON

     BEGIN
          RANGE
          FILTER
          CALC
          ADJUST
          VELOCITY
          SPEED
          ?SWITCH
     UNTIL
          CHAIR-OFF
          RANGING-OFF
   0 UNTIL ;
```

Figure Four
Software operation of electric wheelchair

action and proportional control make it faster to operate than the discrete command characteristics of voice recognition units. In the most general case, the UHCU can map two degrees of head position information (forward/backward and left/right) into an alternate control space.

Users of a modified electric wheelchair equipped with the Ultrasonic Head Control Unit can navigate the chair by tilting their head off the vertical axis. The changing head position is translated by the on-board Forth computer into speed and direction signals for the electric motors on the chair. Users thus direct the motion of the chair with their head. To travel forward, one wquld move the head forward of its normal relaxed vertical position. Similar movements perform motion in the remaining three directions: left pivot, right pivot and backwards. Since this system accepts combinations and degrees of these motions, a smooth right turn can be accomplished by positioning the head slightly forward and to the right. In effect, the user's head has become a substitute for the joystick control found on some electric wheelchairs.

In fact, the signals that the UHCU produce exactly mimic those produced by a joystick. In wheelchair applications, the UHCU can be simply plugged into the motor controller instead of the joystick. In this manner, no modifications to the motor controller need to be made; the UHCU becomes an electric module providing head position control. From the perspective of the motor controller, the UHCU generates signals that emulate those produced by the joystick from which it was originally made to operate. Other devices that normally use a joystick or switch closure as the human input mechanism can use an adapted Ultrasonic Head Control instead.

In actual operation, the wheelchair system performs quite satisfactorily. After a minimum amount of training and practice (usually under one hour), its control can be mastered by even the most severely injured individuals who still retain good head position control. The head tilting required is so slight, only an inch or two, that observers frequently can not deduce the method of control. Since the UHCUonly responds to head tilts, the user can freely move the eyes or rotate the head without affecting the navigation path. In this manner, one can watch for automobiles at intersections or converse with others while traveling.

In this instance, the UHCU is completely transparent to the operator. The existence of any computer hardware or Forth software is not apparent to the user. One "test pilot" commented that the system was so "high tech" that it appeared "low tech."

Other applications of the Ultrasonic Head Control Unit are being pursued. The head control unit of a robotic arm system and operation of a communication system will permit disable individuals more efficient manipulation of objects and words.

References

1. *Jib Ray Forth*, distributed by Bob Stratton, Applied Digital Systems, 805/257-0244.

Simple Data Transfer Protocol

Keith Ericson and Dennis Feucht
Beaverton, Oregon

Data communication:
Simple or Complex

One of the most readily available data communications media is the RS232C Serial Asynchronous Protocol (referred to here as RS232). It can be used to transfer any data, be it ASCII only (text), binary (object code) or Forth screens, between quite different computing systems. The need for such data transfer frequently arises when persons such as local FIG members with different systems want to exchange data (such as Forth programs). The problem is to coordinate the sending station and receiving station to ensure reliable communication: this is the task of a *data transfer protocol.*

A simpler protocol is desirable for at least two reasons:
1) If the destination system has no compatible communication protocol, the receive words will probably be typed by hand. The decreased amount of typing is desirable.
2) Often one must understand the protocol to implement the system-dependent details. The concepts of the protocol presented here are straight-forward and readily grasped.

As a simple protocol, this method lacks error-checking. However, for relatively reliable channels (which is often the case), this is not a serious omission. In our experience, only rarely has a screen of bad data needed to be re-sent.

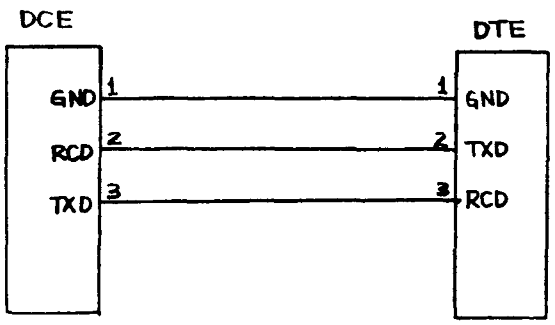


Figure One
Simple RS-232 Connection

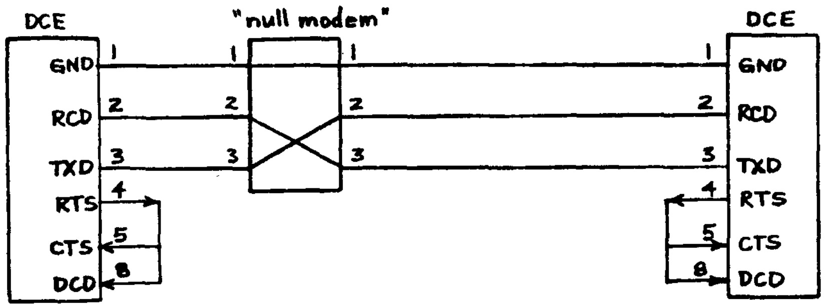


Figure Two
Complete RS-232 Connection for Simple Communications

Theory of Operation

Before any data can be transferred, the transmitting function (the Forth word **XMT**) and the receiving function (**RCV**) must be synchronized. An important consideration in the design of this protocol is that either **XMT** or **RCV** could be the first to begin executing; whichever starts first will wait for a signal from the other before proceeding to transfer data.

The required synchronization scheme, effected by the word **SYNC**, is the first function executed by both the transmit and receive stations. It works by exchanging specific control characters between the two stations. Therefore, the serial-port receive buffers must first be cleared to ensure that whatever is in the buffer is not erroneously interpreted as one of the synchronization characters. **SYNC** clears the receive buffers by sending two ASCII nul (hex 0) characters. **SYNC** causes the receive and transmit stations then to enter a loop which sends enq characters (hex 5) while simultaneously looking for incoming enqs. As soon as an enq is received, a single ack character (hex 6) is sent, and the station then begins waiting to receive a return ack. When both stations have received their ack, the stations are synchronized and **SYNC** exits.

Once **XMT** and **RCV** are synchronized, data transfer commences. **XMT** immediately enters the word **ENQ** which sends a single enq and waits for an ack from **RCV**. **RCV**, meanwhile, calls **BLOCK** to assign a memory buffer for the incoming data. When **BLOCK** is completed, which could take some time if disk access is required, the word **ACK** is then executed. **ACK**'s function is to respond to an enq received from the other station by sending an ack. (If necessary, **ACK** waits for the enq to arrive.) Transmission of the ack by the receive station indicates it is

```
SCREEN 1
        ( forth screen transfer utility - fig-Forth )
        : SEND-BLOCK ( block-addr --- )
            DUP B/BUF + SWAP
            BEGIN DUP C@ XOUT 1+ 2DUP =
            UNTIL 2DROP ;

        : TAKE-BLOCK ( block-addr --- )
            DUP B/BUF + SWAP
            BEGIN XIN OVER C! 1+ 2DUP =
            UNTIL 2DROP UPDATE ;

        : SYNC TXCLR
            BEGIN 5 XOUT ?XIN
                  IF XIN 5 = ELSE 0 THEN
            UNTIL 6 XOUT
            BEGIN XIN 6 = UNTIL ;

SCREEN 2
        ( forth screen transfer utility - fig-Forth )
        : XMT      ( first-scr# last-scr# --- )
            SYNC  1+ SWAP
            DO   ENQ 2 XOUT I BLOCK SEND-BLOCK
            LOOP ENQ 4 ( eot ) XOUT ;

        : RCV     ( first-scr# --- )
            SYNC
            BEGIN DUP BLOCK ACK XIN 2 ( stx ) OVER =
                  IF   DROP TAKE-BLOCK  0
                  ELSE 4 ( eot ) =
                       IF   DROP FLUSH
                       THEN 1
                  THEN SWAP 1+ SWAP
            UNTIL DROP ;



SCREEN 3
        ( communication primitives - IBM-PC )
        HEX
        3F8 CONSTANT PORT       3FD CONSTANT STATUS

        : ?XOUT    STATUS PC@ 20 AND ;
        : ?XIN     STATUS PC@ 01 AND ;

        : XOUT     BEGIN ?XOUT UNTIL PORT PC! ; ( WAIT, SEND A CHAR )
        : XIN      BEGIN ?XIN  UNTIL PORT PC@ ; ( WAIT, GET A CHAR )

        : TXCLR 0 XOUT 0 XOUT ;

        : ENQ BEGIN 5 ( enq ) XOUT XIN 6 ( ack ) = UNTIL ;
        : ACK BEGIN XIN 5 ( enq ) = UNTIL 6 ( ack ) XOUT ;

        DECIMAL
```

ready to receive data. When the transmit station receives the ack, the wait loop in **ENQ** will be exited. If another block of data is to be sent, **XMT** sends an **STX** character (hex 2). **RCV** then expects to receive **B/BUF** bytes of data, which immediately follow. If, on the other hand, no more data is to be sent, **XMT** sends an EOT character (hex 4), causing **RCV** to save the contents of the memory buffers onto disk and exit.

The Listings

Screens 1 and 2 list the intermediate-level functions (**SEND-BLOCK** and **TAKE-BLOCK**) and the high-level functions **XMT** and **RCV**. These functions are identical for all the systems — IBM PC, Apple II and Rockwell RSC-Forth. fig-FORTH and Forth-79 implementations, to be useful for the entire address space, use the **BEGIN-UNTIL** construct to define words **SEND-BLOCK** and **TAKE-BLOCK**. (In Forth-83 systems these could be implemented with the simpler **DO-LOOP** construct, with the loop index used as as the block-address). Screens 3, 4 and 5 include system specific serial-communication primitives for the systems implemented: IBM PC running Laboratory Microsystems' PC-Forth; Apple II running MicroMotion's Forth-79; and the Rockwell R65F11 single-chip Forth computer running RSC-Forth (version 1.5). Screen 6 is a terminal emulator for the Apple II and is explained below (see notes on Apple listing).

Serial Communication Primitives

On the Rockwell RSC-Forth system a serial communication port is provided, but both the IBM PC and the Apple II require the addition of a serial communication card. Each system will require its own set of serial communications primitives, determined by both hardware and software considerations. A 6850 ACIA will require different code than will an 8251; machines that access serial communication functions through an operating system will require different code than "direct access" systems. The primitives for your system will very likely be different.

The word **?XKEY** differs from **?KEY** in that it checks for the presence of an input from the keyboard without affecting the keyboard strobe bit; **KEY** is then used to read the detected character.

For each computer system, the communication-port primitives must be supplied. They are the following:

XINIT performs communication initialization. This may include, but is not necessarily limited to, asserting RTS and setting of baud rate, number of bits, parity and number of stop bits. (This is actually used outside of the data-transfer functions in preparing the computer to correctly interface to the RS232 port.)

?XIN determines whether or not a character has been received, and returns a true flag if so. Note that a character is not waited for, nor is it retrieved if one has arrived.

?XOUT determines whether or not the previous character has been transmitted, and returns a true flag if so.

XIN waits for a character to be received, then fetches it.

XOUT waits for previous character to be transmitted, then stores the current character for its transmission.

RS232 Link — Data and Control Signals

For the simplest RS232 link, the two stations are connected together as shown in figure one. Notice that pins 2 and 3 are interchanged in the RS232 connectors. The DCE (Data Communications Equipment) connector is normally used for the computer or modem and the DTE (Data Terminal Equipment) connector is used for the terminal. When both stations have DCE connectors, however, an adapter (called a "null modem") is required to interchange pins 2 and 3, the Transmitted Data (TxD) and the Received Data (RxD) pins.

RS232 has several control lines, at least two of which must be considered here. The receiver will be enabled only if the DCD line is asserted, and the transmitter will be enabled only if the CTS line is asserted. A loop connection at each station, in which the local RTS (Request To Send) signal is usedto assert both the local CTS and DCD signals, is shown in figure two. (Alternatively, the

```
SCREEN 4
        ( communication primitives    APPLE ][  )
        HEX      3 CONSTANT COMSLOT#
        COMSLOT# 10 * C080 + CONSTANT XSTATUS
                  XSTATUS 1+ CONSTANT XDATA

        : XINIT 3 XSTATUS C! 9 XSTATUS C! ;

        : ?XIN XSTATUS C@ 1 AND ;

        : XIN BEGIN ?XIN UNTIL XDATA C@ ;

        : ?XOUT XSTATUS C@ 2 AND ;

        : XOUT BEGIN ?XOUT UNTIL XDATA C! ;
        DECIMAL

SCREEN 5
        ( communication primitives - RSC-FORTH )
        HEX

        : ?XOUT SCSR C@ 40 AND ;
        : ?XIN SCSR C@ 01 AND ;

        : XOUT BEGIN ?XOUT UNTIL SCDR C! ; ( wait, send a char )
        : XIN BEGIN ?XIN UNTIL SCDR C@ ; ( wait, get a char )

        : TXCLR 0 XOUT 0 XOUT ;

        : ENQ BEGIN 5 ( enq ) XOUT XIN 6 ( ack ) = UNTIL ;
        : ACK BEGIN XIN 5 ( enq ) = UNTIL 6 ( ack ) XOUT ;

        DECIMAL

SCREEN 6
        ( APPLE TERMINAL EMULATOR - USE NULL MODEM )
        HEX CODE ?XKEY C000 LDA, 80 # AND, PHA, TYA, PUSH JMP, END-CODE
        DECIMAL
        : TERMINAL XINIT
             BEGIN 1 ?XKEY
                  IF KEY DUP 24 ( CAN) =
                       IF DROP 1-
                       ELSE XOUT
                       THEN
                  THEN
             WHILE ?XIN
                  IF XIN DUP 10 ( LF) =
                       IF DROP ELSE EMIT THEN
                  THEN
             REPEAT ;
```

CTS and DTR signal lines could be connected to the hardware's positive supply, +5 to +12 volts, typically, through a 1k- to 4.7k-ohm resistor.)

Notes on Apple Listing

A terminal emulator is included with the Apple listing. Its use is to verify that the RS232 link functions properly before attempting data transfer. To use it, invoke **TERMINAL**. This word initializes the communication port, then checks the keyboard (through **?KEY**) to determine if a key has been pressed. If so, it calls **KEY** to get the character. If it is a can character (hex 18, a control-X), it exits **TERMINAL**; otherwise, the character is sent by **XOUT**. Then the serial port is checked for an incoming character. If one is present, it is fetched by **XIN**, but ignored if it is a lf character (hex 10, or line feed); otherwise it is **EMIT**ed to the screen. The loop is then repeated.

The word **?XKEY** differs from **?KEY** in that it checks for the presence of an input from the keyboard without affecting the keyboard strobe bit; **KEY** is then used to read the detected character.

On the first screen of the Apple listing, the serial communication words assume a 6850 ACIA located in I/O slot number three. **XINIT** will initialize the port; the ACIA control register (which is located at **XSTATUS**, the address of both the read-only status register and the write-only control register) is initialized so that RTS remains asserted, thus enabling communications via the local loops in figure two.

Integer *(Continued from page 19)*

SAVE, **RESTORE** and **REPLACE** when implemented in native code make programs run faster, compile smaller and read more clearly. Also, these words allow definitions which use variables to be re-entrant. This is important when multi-tasking or recursion is desired.

Bibliography

McNeil, Michael, "The TO Variable," *1980 FORML Conference Proceedings.*

Nieuwenhuijzen, Hans, "Standard Forth to TO-Forth," *1980 FORML Conference Proceedings.*

Bartholdi, Paul, "The TO Solution," *Forth Dimensions,* Volume I, Number 4,5.

Inner Access holds the key to your software solutions

When in-house staff can't solve the problem, make us a part of your team. As specialists in custom designed software, we have the know-how to handle your application from start to finish.

Call us for some straight talk about:

- Process Control
- Automated Design
- Database Management
- System Software & Utilities
- Engineering
- Scientific Applications
- Turn Key Systems

Inner Access Corporation
P.O. Box 888, Belmont, CA 94002
PHONE (415) 591-8295

THE FORTH SOURCE™

MVP-FORTH

Stable - Transportable - Public Domain - Tools

You need two primary features in a software development package . . . a stable operating system and the ability to move programs easily and quickly to a variety of computers. MVP-FORTH gives you both these features and many extras. This public domain product includes an editor, FORTH assembler, tools, utilities and the vocabulary for the best selling book "Starting FORTH". The Programmer's Kit provides a complete FORTH for a number of computers. Other MVP-FORTH products will simplify the development of your applications.

MVP Books - A Series

- ☐ **Volume 1,** *All about FORTH* by Haydon. MVP-FORTH glossary with cross references to fig-FORTH, *Starting FORTH* and FORTH-79 Standard. 2nd Ed. $25
- ☐ **Volume 2,** *MVP-FORTH Assembly Source Code.* Includes CP/M®, IBM-PC®, and APPLE® listing for kernel $20
- ☐ **Volume 3,** *Floating Point Glossary* by Springer $10
- ☐ **Volume 4,** *Expert System* with source code by Park $25
- ☐ **Volume 5,** *File Management System* with interrupt security by Moreton $25

MVP-FORTH Software - A Transportable FORTH

- ☐ **MVP-FORTH Programmer's Kit** including disk, documentation, Volumes 1 & 2 of MVP-FORTH Series (*All About FORTH*, 2nd Ed. & Assembly Source Code), and *Starting FORTH*. Specify ☐ CP/M, ☐ CP/M 86, ☐ CP/M+, ☐ APPLE, ☐ IBM PC, ☐ MS-DOS, ☐ Osborne, ☐ Kaypro, ☐ H89/Z89, ☐ Z100, ☐ TI-PC, ☐ MicroDecisions, ☐ Northstar, NEW ☐ Compupro, ☐ Cromenco, ☐ DEC Rainbow, ☐ NEC 8201, ☐ TRS-80/100 $150
- ☐ **MVP-FORTH Cross Compiler** for CP/M Programmer's Kit. Generates headerless code for ROM or target CPU $300
- ☐ **MVP-FORTH Meta Compiler** for CP/M Programmer's kit. Use for applicatons on CP/M based computer. Includes public domain source $150
- ☐ **MVP-FORTH Fast Floating Point** Includes 9511 math chip on board with disks, documentation and enhanced virtual MVP-FORTH for Apple II, II+, and IIe. $450
- ☐ **MVP-FORTH Programming Aids** for CP/M, IBM or APPLE Programmer's Kit. Extremely useful tool for decompiling, callfinding, and translating. $200
- ☐ **MVP-FORTH PADS (Professional Application Development System)** for IBM PC, XT or PCjr or Apple II, II+ or IIe. An integrated system for customizing your FORTH programs and applications. The editor includes a bi-directional string search and is a word processor specially designed for fast development. PADS has almost triple the compile speed of most FORTH's and provides fast debugging techniques. Minimum size target systems are easy with or without heads. Virtual overlays can be compiled in object code. PADS is a true professional development system. Specify Computer. $500
- NEW ☐ **MVP-FORTH Floating Point & Matrix Math** for IBM or Apple $85
- NEW ☐ **MVP-FORTH Graphics Extension** for IBM or Apple $65
- NEW ☐ **MVP-FORTH MS-DOS** file interface for IBM PC PADS $80
- NEW ☐ **MVP-FORTH Expert System** for development of knowledge-based programs for Apple, IBM, or CP/M. $100

FORTH CROSS COMPILERS Allow extending, modifying and compiling for speed and memory savings, can also produce ROMable code. Specify CP/M, 8086, 68000, IBM, Z80, or Apple II, II+ $300

FORTH COMPUTER

- ☐ **Jupiter Ace** $150

Ordering Information: Check, Money Order (payable to MOUNTAIN VIEW PRESS, INC.), VISA, MasterCard, American Express. COD's $5 extra. Minimum order $15. No billing or unpaid PO's. California residents add sales tax. Shipping costs in US included in price. Foreign orders, pay in US funds on US bank, include for handling and shipping by Air: $5 for each item under $25, $10 for each item between $25 and $99 and $20 for each item over $100. All prices and products subject to change or withdrawal without notice. Single system and/or single user license agreement required on some products.

FORTH DISKS

FORTH with editor, assembler, and manual.

- NEW ☐ **APPLE** by MM, 83 $100
- ☐ **APPLE** by Kuntze $90
- ☐ **ATARI®** valFORTH $60
- NEW ☐ **CP/M®** by MM, 83 $100
- ☐ **HP-85** by Lange $90
- ☐ **HP-75** by Cassady $150
- NEW ☐ **IBM-PC®** by LM, 83 $100
- ☐ **NOVA** by CCI 8" DS/DD $175
- ☐ **Z80** by LM, 83 NEW $100
- ☐ **8086/88** by LM, 83 NEW $100
- ☐ **68000** by LM, 83 NEW $250
- ☐ **VIC FORTH** by HES, VIC20 cartridge $50
- ☐ **C64** by HES Commodore 64 cartridge $60
- ☐ **Timex** by HW $25

Enhanced FORTH with: F-Floating Point, G-Graphics, T-Tutorial, S-Stand Alone, M-Math Chip Support, MT-Multi-Tasking, X-Other Extras, 79-FORTH-79, 83-FORTH-83.

- NEW ☐ **APPLE** by MM, F, G, & 83 $180
- ☐ **ATARI** by PNS, F,G, & X. $90
- NEW ☐ **CP/M** by MM, F & 83 $140
- ☐ **Apple, GraFORTH** by I $75
- ☐ **Multi-Tasking FORTH** by SL, CP/M, X & 79 $395
- ☐ **TRS-80/I or III** by MMS F, X, & 79 $130
- ☐ **Timex** by FD, tape G,X, & 79 $45
- ☐ **Victor 9000** by DE,G,X $150
- ☐ **C64** by ParSec. MVP, F, 79, NEW G & X $96
- ☐ **FDOS** for Atari FORTH's $40
- ☐ **Extensions** for LM Specify IBM, Z80, or 8086
  - ☐ Software Floating Point $100
  - ☐ 8087 Support (IBM-PC or 8086) $100
  - ☐ 9511 Support (Z80 or 8086) $100
  - ☐ Color Graphics (IBM-PC) $100
  - ☐ Data Base Management $200
- ☐ **fig-FORTH Programming Aids** for decompiling, callfinding, and translating. CP/M, IBM-PC, Z80, or Apple $200

FORTH MANUALS, GUIDES & DOCUMENTS

- ☐ **ALL ABOUT FORTH** by Haydon. See above. $25
- ☐ **FORTH Encyclopedia** by Derick & Baker $25
- NEW ☐ **The Complete FORTH** by Winfield $16
- ☐ **Understanding FORTH** by Reymann $3
- ☐ **FORTH Fundamentals,** Vol. I by McCabe $16
- NEW ☐ **FORTH Fundamentals,** Vol. II by McCabe $13
- NEW ☐ **FORTH Tools, Vol.1** by Anderson & Tracy $20
- ☐ **Beginning FORTH** by Chirlian $17
- ☐ **FORTH Encyclopedia Pocket Guide** $7
- ☐ **And So FORTH** by Huang. A college level text. $25
- ☐ **FORTH Programming** by Scanlon $17
- ☐ **FORTH on the ATARI** by E. Floegel $8
- ☐ **Starting FORTH** by Brodie. Best instructional manual available. (soft cover) $18
- ☐ **Starting FORTH** (hard cover) $23
- ☐ **68000** fig-Forth with assembler $20
- ☐ **Jupiter** ACE Manual by Vickers $15
- ☐ **1980 FORML Proc.** $25
- ☐ **1981 FORML Proc** 2 Vol $40
- ☐ **1982 FORML Proc.** $25
- ☐ **1981 Rochester FORTH Proc.** $25
- ☐ **1982 Rochester FORTH Proc.** $25
- ☐ **1983 Rochester FORTH Proc.** $25
- ☐ **A Bibliography of FORTH References, 1st. Ed.** $15
- **The Journal of FORTH Application & Research**
  - ☐ **Vol. 1, No. 1** $20
  - NEW ☐ **Vol. 1, No. 2** $20
- ☐ **A FORTH Primer** $25
- ☐ **Threaded Interpretive Languages** $23
- ☐ **METAFORTH** by Cassady $30
- ☐ **Systems Guide to fig-FORTH** $25
- ☐ **Invitation to FORTH** $20
- ☐ **PDP-11 User Man.** $20
- ☐ **FORTH-83 Standard** $15
- ☐ **FORTH-79 Standard** $15
- ☐ **FORTH-79 Standard Conversion** $10
- ☐ **Tiny Pascal fig-FORTH** $10
- ☐ **NOVA fig-FORTH** by CCI Source Listing $25
- ☐ **NOVA** by CCI User's Manual $25
- ☐ **Installation Manual for fig-FORTH,** $15

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

☐ 1802	☐ 6502	☐ 6800	☐ AlphaMicro	
☐ 8080	☐ 8086/88	☐ 9900	☐ APPLE II	
☐ PACE	☐ 6809	☐ NOVA	☐ PDP-11/LSI-11	
☐ 68000	☐ Eclipse	☐ VAX	☐ Z80	☐ IBM NEW

MOUNTAIN VIEW PRESS, INC.

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

Ask the Doctor

Forth and the AIM-65

William F. Ragsdale
Hayward, California

"Ask the Doctor" is Forth Dimensions' *health maintenance organization devoted to helping you use and understand Forth. Questions about problems you have, references you need or contemporary techniques are most appropriate. When needed, your good doctor will call in specialists. Published letters will receive a pre-print of the column as a direct reply.*

Questions on locating products, such as "Where can I get Forth for my Pet, Apricot, Apple, Cray, Amdahl, etc." may be better answered by reading the advertisements in your favorite magazine. We have found that if the vendor is unwilling to advertise his product, the likelihood is great that documentation and support will also be limited.

Morning rounds having been completed, we find the doctor seated at his terminal, contemplating the latest correspondence from readers.

Professor Jose J. Ruz-Ortiz of the University of Madrid, Spain writes, "I am very interested about the use of Forth in process control. In our student laboratory in the Department of Computer Science and Automatic Control, we have AIM-65 systems for student projects. I would be grateful for appropriate bibliographic references."

Rx: I assume you have the Forth ROMs for the AIM-65 offered by Rockwell. This version of fig-FORTH for the 6502 was supplied by the good doctor to John Baumgarner and Dave Boulton, who customized it to the display and I/O of the AIM. This machine offers a full keyboard, one-line display and a selection of peripherals. Rockwell documentation is excellent.

The volume *A Bibliography of Forth References* edited by David Hoffert (published by the Institute for Applied Forth Research, Inc., 70 Elmwood Ave., Rochester, New York 14611) surveys the entirety of Forth literature. This volume is an invaluable aid to the doctor. It should be part of the library of any serious Forth user. For example, the subject listing on applications cites fifty-two articles with subject matter as diverse as parking lot control and radar control.

The subject of Forth on the AIM-65 reminds the good doctor of an interesting event of international note. About 1979, an excursion was made to China (PRC) by Ray Dessey of Virginia Polytechnic, who took with him Forth and an AIM computer. Dr. Dessey lectured in China on laboratory automation, gave many Chinese students their first view of a personal computer (and Forth!) and, upon his return, related his experience to the Northern California FIG Chapter.

Glenn Mitchell of Bedford, Indiana is using the HES cartridge of fig-FORTH (written by FIG's own Tom Zimmer) on the Commodore 64. Glenn's letter states, "My goal is to program real-time analysis for audio systems. If there is any written material on this subject or tools available (fast Fourier transforms, etc.) that might help, please let me know.

In a similar vein (ouch!), we have a query from Ken Fasano of Hixson, Texas asking, "What information can you offer on using Forth as a tool for automated music composition with microcomputers. I am interested in the researches of composers such as Xenakis and Koenig being made available to the individual microcomputerist."

Rx: A definitive article on the subject of a Forth interface to music is by Kim Harris (FIG's secretary) and Jeff Morris (Bell Telephone Labs) in the *1983 Rochester Forth Conference Proceedings*, pp. 43-66. Although the instrument was a special-purpose processor, the Forth techniques are quite general. This is a follow-up to the paper Jeff presented at the 1982 Rochester Forth Conference.

Another item is "Music Generation in Forth" by Michael Burton, *Forth Dimensions* Volume III, Number 2, pp. 54-56. This article might be particularly interesting, as it presents an interface to the General Instrument programmable sound generator AY3-8910. This interface could serve as a template to be modified to match the Commodore chip. Scores are given in Forth source code for "Red River Valley" and "Jesus Christ, Superstar."

A final paper is "Fast Fourier Transform in Forth," Dr. Hans Nieuwenhuijzen, *1982 Rochester Forth Conference Proceedings*, pp. 241-246. Dr. Nieuwenhuijzen is a contributor to the Forth standards effort and has authored numerous papers on Forth. This work is based on the Cooley-Tukey algorithm as coded by J. Brault at Kitt Peak National Observatory and requires eight floating-point primitives.

The FIG-Tree dial-in data base on Forth is a bulletin board system operated by your faithful Forth practitioner (call 415-538-3580). Our next question was found on the FIG-Tree, contributed anonymously. The user asks, "Logo offers the ability to later modify the definition of an already-compiled procedure. It would be nice if Forth could offer such an ability. For example, if the definition

```
: CIRCUMFERENCE DIAMETER
  PI-TIMES ;
```

uses a definition for **PI-TIMES** as

```
: PI-TIMES  22 7 */ ;
```

then we could later refine the precision by re-defining **PI-TIMES** as

```
: PI-TIMES  355 113 */ ;
```

Continued on page 41

Techniques Tutorial

Part I

Debugging Techniques

Henry Laxen
Berkeley, California

I realize that most of you will have little need for this article since, by utilizing the modular approach of programming in Forth, most of your applications simply work bug-free the first time. Occasionally, however — perhaps when you are forced to modify the code of someone who does not adhere to your strictly imposed self-discipline and exemplary style — you may find some of these techniques valuable. In this article I will try to adhere to relatively machine-independent approaches to debugging. There are far more general and sophisticated techniques available to you once you allow yourself to be very machine specific, such as selective tracing of previously defined definitions and even monitoring individual locations in memory and trapping them when they are modified per your specification. We will cover those techniques in my next article.

One of the proclaimed virtues of Forth is its lack of any run-time overhead associated with error checking. This is both a blessing and a curse. There is certainly much comfort to be had by knowing that the machine is trying to help you catch your own mistakes. The longer your program runs without the machine detecting an error, the higher your confidence level in your application code. Unfortunately, the price you pay is often exorbitant and most other languages do not give the programmer the choice of whether or not he is willing to pay this price. After all, the compiler writer is smarter than you and knows what is best for you, right? The Forth approach is to put the full responsibility for run-time error checking on the programmer. While this is probably an overwhelming burden for the novice, after reading this article you will find that, maybe, it's not so difficult after all.

There are two basic items that need to be implemented for our debugging tool kit. The first is to decide on what should be done when a run-time error is detected. The second is to figure out how to detect run-time errors. Before I continue, I should mention that all of the code listed in this article is written in F83, the public-domain implementation of Forth by Mike Perry and myself. F83 is based on the 83-Standard, and if your system is either fig-FORTH or Forth-79 you will need to modify portions of it.

Now then, suppose we have just detected a run-time error of some type. What now? Well, there are two very important pieces of information we are generally interested in: what was on the parameter stack and where we were in our program at the time the error occurred. Most Forth systems contain, as a tool that helps us with the contents of the parameter stack, the word **.S** which is a non-destructive stack printing word. The code in figure one illustrates the implementation of **.S** for a Forth-83 system. We first check if the number of items on the stack is negative. It is quite rare that we have 32,768 legitimate items on the stack, and even rarer that we want to display them; so negative depths are treated as errors and generate the "Stack Underflow" message. If the **DEPTH** was zero, then the stack was empty and we tell the user so. Otherwise, we print out each item on the stack as an unsigned number right-justified in a field of width seven and with a trailing space. The **KEY? ?LEAVE** allows us to exit the definition if there is more on the stack than we care to see. Since we used **PICK** to grab the items, we leave the contents of the stack unchanged. Thus **.S** is an ideal run-time debugging tool since it displays information on the screen without disturbing the state of the Forth system. You can simply include a **.S** anywhere in your code when you are debugging without disturbing anything. I suggest you do so when you suspect a word is being passed incorrect data.

The second piece of information we are interested in is what exactly was running at the time when the detected error occurred. The word commonly delegated to this task is called **UNRAVEL** in most systems since its purpose is to unravel the nesting structure and display it on the screen. Let us briefly review how Forth nesting works.

When the Forth inner interpreter executes a high-level (i.e. defined by :) word, the run-time action is to save the current value of the interpretive pointer (IP) on the return stack and change the value of the IP to point to the parameter field of the new definition. If that sentence meant something to you, then you will realize that in order to **UNRAVEL** the nesting structure, all you need to do is peel off items from the return stack and display the name associated with each of them. The simplest version of **UNRAVEL**, which is present in many systems, is listed in figure two. **RP@** returns the address of the top of the return stack. **RP0** is a variable that contains the initial address of the return stack, when it is empty. Thus, while these two values are unequal, we pop a number off the return stack, back it up by two (if,as most, you are running a post-incrementing system), fetch the contents of that location (which should be the code field of the word currently executing), get to the name field with **>NAME** and print it with **.ID**. Once the return stack is empty, we exit the **BEGIN WHILE REPEAT** loop and execute **QUIT**, which ends what we were doing and calls the **INTERPRET**er. This version of **UNRAVEL** works well and can be used whenever we detect a fatal error. However, it can be improved.

The problem which you will soon discover with the above version of **UNRAVEL** is that often it will print strings that are obviously not names in your Forth system. Something is wrong somewhere. Unfortunately, the return stack in Forth is not used exclusively for holding return addresses. In most Forth systems, the return stack is also used to hold **DO LOOP** limits and indices. Obviously, when a random **DO LOOP** limit or index is converted to a **NAME** field address, the old garbage-in garbage-out syndrome takes place. Furthermore, many programmers have the unfortunate habit of using the return stack as a temporary data storage area. When they do this, they are generally not putting a valid code field onto the return stack. Thus, it would be nice if we could distinguish between IP addresses that are saved on the return stack and other data, such as that put there by **DO LOOP**s or bad programmers. Fortunately this, too, is easy.

There are only two types of definitions in Forth that cause nesting to take place. The obvious kind is colon definitions. One property shared by all of these is that the contents of their code field points to the same piece of code. Thus, for any word defined by : the following will yield the same result: ' <word> **?** since the code fields all point to the run-time code for :. This gives us a way of testing whether or not a value on the return stack is really a saved IP value that was pushed there by the run-time action of a colon definition.

The other type of definition that causes nesting to take place is, of course, the run-time portions of high-level defining words, i.e. our old friends **CREATE** and **DOES>**. If we are executing the **DOES>** portion of a word then the run time for **DOES>** also pushes the IP onto the return stack, just like :. In addition, it also pushes the address of the parameter field onto the parameter stack, but we can ignore that. The rule for detecting **DOES>** words on the return stack is slightly different. If we fetch the contents of the value on the return stack, then that address must contain a CALL or JSR instruction to the run-time code for **DOES>**. Because different processors store the target address of the CALL instruction in different ways, we can't really verify that the CALL instruction is pointing at the same code in a very machine-independent way. Thus, all we can really do is check for the existence of the CALL instruction itself. The code in figure three illustrates how to do this and also how to modify the code in figure two to take into account our new refinement. Thus, our latest definition of **UNRAVEL** will print the nesting structure completely and will only print valid

```
: .S    (S -- )
   DEPTH DUP 0< ABORT" Stack Underflow"
   DUP IF    0 DO   DEPTH I - 1- PICK   7 U.R SPACE
             KEY? ?LEAVE   LOOP
   ELSE    ." Empty"   THEN    ;
```

Figure One

```
: UNRAVEL     (S -- )
   BEGIN   RP@ RP0 @ <> WHILE    R> 2- @ >NAME .ID    REPEAT
   QUIT   ;
```

Figure Two

```
: HIGH?    (S addr -- f )
   DUP  @     [ ' : @ ]          LITERAL =
   SWAP @ C@ [ ' FORTH @ C@ ] LITERAL = OR    ;

: UNRAVEL    (S -- )
   BEGIN   RP@ RPO @ <> WHILE
      R> DUP   2- @   DUP HIGH? IF   >NAME .ID DROP
      ELSE   DROP U.   THEN
   REPEAT   QUIT   ;
```

Figure Three

```
: ?ENOUGH    (S n -- )
   DEPTH 1- > IF   CR ." Not enough parameters " UNRAVEL    THEN ;

: ?EXACTLY    (S n -- )
   DEPTH 2DUP 1- <> IF
      CR ." Wrong # of parameters: " . ." actual, and "
      . ." expected "   UNRAVEL
   ELSE   2DROP   THEN   ;
```

Figure Four

```
: DROP    1 ?ENOUGH DROP    ;          : 2DROP 2 ?ENOUGH 2DROP   ;
: DUP     1 ?ENOUGH DUP     ;          : 2DUP  2 ?ENOUGH 2DUP    ;
: OVER    2 ?ENOUGH OVER    ;          : NIP   2 ?ENOUGH NIP     ;
: ROT     3 ?ENOUGH ROT     ;          : -ROT  3 ?ENOUGH -ROT    ;
: +       2 ?ENOUGH +       ;          : -     2 ?ENOUGH -       ;
: *       2 ?ENOUGH *       ;          : /     2 ?ENOUGH /       ;
: @       1 ?ENOUGH @       ;          : !     2 ?ENOUGH !       ;
```

Figure Five

```
: EXACTLY    (S -- )
   BL WORD    NUMBER DROP    [COMPILE] LITERAL
   COMPILE ?EXACTLY    ; IMMEDIATE

: EXAMPLE    (S n1 n2 -- )
   EXACTLY 2  ." There are 2 numbers on the stack "  SWAP . .  ;

: EXACTLY    (S -- )    ( Make it disappear )
   BL WORD DROP    ; IMMEDIATE
```

Figure Six

```
: MAP    (S addr -- addr' )
   2DUP @ U< IF    2+ SWAP 2* +
   ELSE    CR ." Subscript out of range, max is " ?
      ." tried " .    UNRAVEL    THEN

: ARRAY
   CREATE    (S n -- )    ,
   DOES>     (S n -- addr )    MAP    ;

: CASE:
   CREATE    (S n -- )    ,  ]
   DOES>     (S n -- )    MAP @ EXECUTE    ;
```

Figure Seven

names. Other values present on the return stack will be printed out as unsigned numbers. Thus, by understanding how your **DO LOOP**s work, you can even figure out the loop limit and current index value at the time of the error.

Now that we have the tools we need to get valuable state information when an error is detected, we need to look at when to use error detection. One very simple and powerful technique is to define the words **?ENOUGH** and **?EXACTLY** as in figure four. **?ENOUGH** will give an **UNRAVEL** trace whenever the number of parameters on the parameter stack is less than expected. This is very useful for detecting stack underflows. **?EXACTLY** will give an **UNRAVEL** trace whenever the precise number of parameters on the stack is not present. You can use **?ENOUGH** to re-define all of the stack and arithmetic primitives to check that enough parameters are present on the stack before they attempt their operation. Figure five illustrates this. The use of **?EXACTLY** is very application-dependent. Once you have determined that, during execution of a particular word, there should be exactly n parameters on the stack, you can use **?EXACTLY** to verify this. This is very useful when some unknown word is leaving extra items on the stack. However, it requires you to insert extra code in your application source.

Figure six illustrates one way of minimizing the damage. By defining **EXACTLY** to take a literal number following it, rather than preceding it, we can later define **EXACTLY** to ignore the following number and compile nothing. Thus, the run-time checking can be thrown away or enabled, depending only on the definition of **EXACTLY** and no other source code needs to be changed. Needless to say, simply not loading the code in figure five will remove the run-time stack underflow checking and will also speed up the resulting application.

Other instances of run-time checking are up to you, as only you know your application. If you are using arrays, you can do run-time bounds checking very easily as illustrated in figure seven. The checking can be removed by simply redefining **MAP**. The rest is up to you and your application. As you can see, the amount of code needed to support run-time error checking is extremely small, especially once the primitive **UNRAVEL** has been defined. Only you are the best judge of what should be checked and for what. No compiler writer can do it for you in all cases.

That is enough for now. Next time, we will look at other debugging techniques, particularly those involving knowledge of the internal structure of your Forth system. These will allow us to retroactively debug previously defined words with no additional compilation and no overhead once debugging is completed. Until then, best of luck and may all your bugs be harmless.

Henry Laxen is Vice-President of Research and Development for Paradise Systems, Inc. He worked on the operating system for the Panasonic/Quasar HHC, the world's first hand-held computer. He will soon participate in a tango competition, and has a cat named Sophie who sounds like a bird.

Products and Announcements

Forth Dimensions welcomes press releases and product announcements, as well as reader letters regarding product performance.

November 16-17 are the dates of the **Sixth Annual Forth Convention** and banquet. The convention will meet the needs of Forth enthusiasts, beginners or experienced professionals, with tutorials, exhibits/vendor booths, lectures and discussions. The event is to be held at the Hyatt Palo Alto (California); for information regarding special room rates and exhibit space, call the FIG Hot Line (415-962-8653) or write the Forth Interest Group (P.O. Box 1105, San Carlos, California 94070).

Siggraph 1984 (July 23-27) will include a panel on microcomputer graphics this year; panelists include Chuck Moore, Bill Atkinson and Susan Kare (of Apple Macintosh fame), Scott Kim (miK ttocS?) and others. A **Forth-specific caucus** will be held during the week, as well as demonstrations of QuickDraw and MacPaint, low-resolution typography and real-time animation. There should be plenty of interest to Forth folks who make it to the Minneapolis venue. For details, call Howard Perlmutter at 408-425-8700.

National Semiconductor Corporation has announced the MA2301 **Forth Language Interpreter**. The development system implements MVP-FORTH and is designed to be used with companion products in the MA2000 family, with the NCS800 or with an 8080/Z80-compatible processor. NSC's product announcement states, "FORTH — it's currently the hottest high level computer programming language in the industry."

fig-FORTH is now available in ROM for the **Epson HX-20**. Talbot Microsystems' two CMOS 8K EPROMs contain interpreter, compiler, assembler, string handling and other extensions for HX-20 I/O devices. In addition, a target compiler development system for HX-20 or other 6801/6301 hardware is available for CP/M-80 or CP/M-68K systems. Call 213-376-9941.

TeleForth for the Apple II is Forth-79 and includes a screen editor, macro assembler, high resolution turtle graphics, floating-point and double-precision math, and a DOS 3.3 interface. It is compatible with most eighty-column cards and with a modified DOS such as Diversi-DOS. Source code and cross compiler optional. Call Telekinetics (Nova Scotia) at 902-443-1813.

Audiogenics (U.K.) produces **Forth for the Commodore 64 and VIC-20** computers. A screen compression technique lessens access time on minimum memory VICs. Any screen can be edited, and the editor makes use of the Commodore screen editing features. In the U.S., call Regenics at 714-639-9396.

Doctor *(Continued from page 37)*

Rx: Not to worry: Forth will not let you down in your quest for language parity. At least three methods may be used to change a word's run-time action without re-compiling. Only the last example is recommended by your counsellor, though.

First, you may locate the offending Forth address and patch the compiled value to the execution address of the newer definition. This is called "hot patching," equivalent to doing auto brain surgery. The pity is that the smallest error will turn into an auto lobotomy. This is definitely the last resort, usually reserved for bug fixes.

Secondly, you may alter the code field contents to direct execution of a later definition. This is similar to the **DOES>** of Forth-83. I can't find this method in print.

The third method, and the only one safe enough for daily use, is the Laxen-Perry **DEFER** and **IS** combination. For your example, you would execute

```
DEFER PI-TIMES
: PI*-COARSE  22 7 */ ;
' PI*-COARSE IS PI-TIMES
```

and later you could substitute

```
: PI*-FINE  355 113 */ ;
' PI*-FINE IS PI-TIMES
```

and the later definition would be the run time for **PI-TIMES**. A simplified version of **DEFER** and **IS** using Forth-83 words is

```
: DEFER  CREATE ['] ABORT ,
  DOES @ EXECUTE ;
: IS  ' >BODY ! ;
```

This method is described in Henry Laxen's article on "Self-Defining Words" in *Forth Dimensions*, Volume V, Number 6, pp. 35-36. His earlier article in Volume III, Number 6, pg. 174 on execution vectors has a deeper discussion of the method.

Until next time, I remain yours in practicing Forth.

FIG Chapters

U.S.

• ARIZONA

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

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

• CALIFORNIA

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

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

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

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

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

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

Stockton Chapter
Call Doug Dillon
209/931-2448

• COLORADO

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

• CONNECTICUT

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

• FLORIDA

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

• ILLINOIS

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

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

Rockwell Chicago Chapter
Call Gerard Kusiolek
312/885-8092

• INDIANA

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

• IOWA

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

• KANSAS

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

• MASSACHUSETTS

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

• MINNESOTA

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

• MISSOURI

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

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

• NEVADA

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

• NEW YORK

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

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

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

• OHIO

Athens Chapter
Call Isreal Urieli
614/594-3731

Cleveland Chapter
Call Gary Bergstrom
216/247-2492

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

• OKLAHOMA

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

• OREGON

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

• PENNSYLVANIA

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

• TEXAS

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

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

• VERMONT

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

• VIRGINIA

Norfolk FIG Chapter
Call William Edmonds
804/898-4099

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

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

FOREIGN

• AUSTRALIA

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

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

• BELGIUM

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

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

• CANADA

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

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

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

• COLOMBIA

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

• ENGLAND

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

• FRANCE

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

• GERMANY

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

• IRELAND

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

• ITALY

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

• SWITZERLAND

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

• REPUBLIC OF CHINA

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

SPECIAL GROUPS

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

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

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

FIGGRAPH
Call Howard Pearlmutter
408/425-8700

FORTH INTEREST GROUP
MAIL ORDER

	USA	FOREIGN AIR
☐ Membership in FORTH Interest Group and Volume VI of FORTH DIMENSIONS	$15	$27
☐ Back Volumes of FORTH DIMENSIONS. **Price per each.** ☐ I ☐ II ☐ III ☐ IV ☐ V	$15	$18
☐ fig-FORTH Installation Manual, containing the language model of fig-FORTH, a complete glossary, memory map and installation instructions	$15	$18
☐ Assembly Language Source Listings of fig-FORTH for specific CPU's and machines. The above manual is required for installation. Check appropriate box(es). **Price per each.**	$15	$18

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

☐ "Starting FORTH, by Brodie. BEST book on FORTH. (Paperback)		$18	$22
☐ "Starting FORTH" by Brodie. (Hard Cover)		$23	$28
☐ PROCEEDINGS: FORML (FORTH Modification Conference)			
☐ 1980, $25USA/$35Foreign			
☐ 1981, Two Vol., $40USA/$55Foreign			
☐ 1982, $25USA/$35Foreign			
ROCHESTER FORTH Conference			
☐ 1981, $25USA/$35Foreign			
☐ 1982, $25USA/$35Foreign			
☐ 1983, $25USA/$35Foreign	Total	$______	
☐ STANDARD: ☐ FORTH-79, ☐ FORTH-83. $15USA/$18Foreign EACH.	Total	$______	
☐ Kitt Peak Primer, by Stevens.		$25	$35
☐ MAGAZINES ABOUT FORTH: ☐ BYTE Reprints 8/80-4/81			
☐ Dr Dobb's Jrnl, ☐ 9/81, ☐ 9/82, ☐ 9/83			
☐ Poplar Computing, 9/83 $3.50USA/$5Foreign EACH.	Total	$______	
☐ FIG T-shirts: ☐ Small ☐ Medium ☐ Large ☐ X-Large		$10	$12
☐ Poster, BYTE Cover 8/80, 16"x22"		$ 3	$ 5
☐ FORTH Programmer's Reference Card. If ordered separately, send a stamped, self addressed envelope.		Free	
	TOTAL	$______	

NAME______________________________ MS/APT__________

ORGANIZATION______________________ PHONE(   )__________

ADDRESS____

CITY______________ STATE_____ ZIP__________ COUNTRY__________

VISA#____________________ MASTERCARD#____________________

Card Expiration Date__________

(Minimum of $15.00 on Charge Cards)

Make check or money order in US Funds on US Bank, payable to: FIG. All prices include postage. No purchase orders without check. California residents add sales tax. 10/83

ORDER PHONE NUMBER: (415) 962-8653

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

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

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

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

Address Correction Requested