Volume 8, Number 2
July/August 1986
$4.00

FORTH Dimensions

Forth Resources via Modem

XMODEM Tutorial

On-Line Documentation

Forth Source Formatter

Dual-CFA Definitions

Forth Dimensions
Published by the
Forth Interest Group
Volume VIII, Number 2
July/August 1986
Editor
Marlin Ouverson
Production
Cynthia Lawson Berglund
Typesetting
LARC Computing

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

Subscription to *Forth Dimensions* is free with membership in the Forth Interest Group at $30 per year ($43 foreign air). For membership, change of address and to submit material for publication, the address is: Forth Interest Group, P.O. Box 8231, San Jose, California 95155.

Symbol Table

Simple; introductory tutorials and simple applications of Forth.

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

Advanced; requiring study and a thorough understanding of Forth.

Code and examples conform to Forth-83 standard.

Code and examples conform to Forth-79 standard.

Code and examples conform to fig-FORTH.

Deals with new proposals and modifications to standard Forth systems.

FORTH Dimensions

FEATURES

DEPARTMENTS

Saddled With Benchmarks

Dear Mr. Ouverson:

Now that Mach1 (our Forth compiler) is becoming a rather efficient development system, we are somewhat concerned with the benchmarks that Forths are saddled with when comparing them to other languages. The only sieves we've seen people use must be corrupt versions of those called "BYTE" and "Colburn" sieves. They are not only wrong (there are 1028 primes between zero and 8192, not 1899) but, judging from the stack manipulations, were perhaps rewritten by people who were unfamiliar with, or unfriendly to Forth. Has someone pirated our benchmarks?

When a person truly wants to find prime numbers (and, indeed, our customers are always asking us for the higher primes), he'd like to do it simply, efficiently and correctly. Could you please publish the correct "BYTE" and "Colburn" sieves, and this Forth-83 sieve as ones which do just that?

Thank you,

Terry Noyes
Palo Alto Shipping Co.
Menlo Park, California

```
DECIMAL

8192 CONSTANT size
size SQRT 1+ CONSTANT flicklimit ( if you don't have SQRT, just use 4 / to be safe)
VARIABLE flags size VALLOT

: primes ( --number of primes)          ( does the primes once)
        flags size 01 FILL              ( initialize the array )
        0                               ( prime counter)
        size 2                          ( range of numbers to check )
        DO
          flags I + C@                  ( see if I is a prime)
           IF
           I flicklimit <       ( no need to try and flick flags once
                                  you get past size^0.5, just keep adding
                                  up the primes)
            IF
              I                 ( this is a prime. Used to increment the +loop)
                flags size +  flags I + I + ( range of addresses to tag)
                DO
                 0 I C! DUP       ( flick flags at multiples of I)
                +LOOP
              DROP       ( drop the I that was used for +LOOP)
             THEN
            1+           ( increment the prime counter )
           THEN
         LOOP ;
```

How it works: Initialize an array to all ones. Start with the first prime number, which is two. Clear all bytes in the flags array that are multiples of two. Then do it for the threes. The fourth byte in "flags" is a zero, so skip to five. Since four is non-prime, all of its multiples were handled by another number, two in this case. You only have to continue this process up to the integer square root of the array size, since in this case 91*91 is not in the array and the 91*90 spot was already zeroed by a previous prime number. If you dump the array, you'll see that bytes 2, 3, 5, 7, 11, 13, etc., are marked with ones, and the others have been zeroed. (While this program runs, it counts all the prime numbers it finds.)

Public-Domain Floating Point and Double/Quad Precision

Dear Marlin,

Forth has gone too long without a complete, standard wordset for extended arithmetic functions! Instead of continually reinventing the wheel with hardware-dependent math packages and spending time writing yet another version of **D***, the Forth community needs to expend its energy addressing new horizons.

In order to foster the rapid growth of a standard, I am placing my book, *MVP-FORTH Integer and Floating-Point Math* (MVP book series, volume three), in the public domain. This book contains a complete, machine-independent, high-level MVP-FORTH glossary and implementation for thirty-two-bit integer math, sixty-four-bit integer math and thirty-two-bit floating-point math with transcendental functions. The book also includes assembler source code for critical words on popular CPUs, to give execution speeds comparable to other high-level languages.

The math package included with the book has been stable and in active use for several years. The word definitions are similar or identical to many of the other proposed "standards." However, this wordset has the advantages of machine independence and public-domain implementation. The book and source code on disk may be ordered from Mountain View Press.

I think that a standard for Forth arithmetic will only emerge from the evolution of a public-domain implementation in common use. I encourage anyone with comments on, or improvements to the implementation described in my book, to write me in care of Mountain View Press.

Phil Koopman
North Kingstown, Rhode Island

Dictionary Magic

Dear Marlin:

Here is a bit of magic for those who need to generate more dictionary space without reducing the number of options they are loading.

The method requires that there be some space available elsewhere, large enough to be useful. I use what would otherwise be a fifth disk buffer. Four buffers are adequate — so much so that I am considering a reduction to three of them.

Here is an example of how to use the highest-numbered disk buffer for dictionary space.

First, crowd (yes, crowd) as many definitions as possible into a screen. I used all of the system variables and constants. Test free space (**SO @ HERE -** .) and compile. Retest free space to find out how many bytes are required for the compiled screen. For a 1024-byte buffer, I wouldn't accept anything less than 975 bytes.

Second, set up to load the screen as soon after the kernel as possible, followed by all of the other screens which you routinely load. For example, suppose you load screens 4 - 10 before each session, and that the new screen is screen 11. Your loading screen (boot screen?) would contain the following:

```
HERE LIMIT$ @  B/BUF
- DUP  LIMIT$ !  DP !
11 LOAD  DP !  4 10 THRU
```

There are ways to enhance this procedure. In my case, screens 4 - 11 would be coming in as a binary image. To make screen 11 an instant load, I first **MOVE**d the contents of the buffer that screen 11 was compiled into back to screen 11 as a binary image. I then altered the loading screen to reverse the process, **MOVE**ing the image on screen 11 into the buffer area, followed by my **BLOAD** of the remaining options. It works just great.

Sincerely,

Gene Thomas
Little Rock, Arkansas

A Screen in the Dark

Dear Mr. Ouverson,

In writing words to read the clock on my AST board from F83, I ran into all the problems Laughing Water referred to in his letter, "You Screen, I Scream" (VII/2). Adding definitions to the existing code immediately filled up a screen, then what do you do? Also, keeping shadow screens lined up is difficult.

In good Forth programming practice, each word should do one job and do it well. Then the next higher level word does not have to concern itself with the details of how the lower-level word works. This principle of hiding complexity is fundamental to modular programming.

Then there is the Forth editor! It hides nothing. The programmer must keep track of screen numbers, line numbers, line length and a host of other items. If a screen is full and an additional definition is needed, the screens must be moved manually. Any reasonable text editor will insert text anywhere in a file, moving the rest down to make room. Also, my screen is eighty columns wide and it is all too easy to enter a definition that is sixty-five characters long and lose the last word.

I propose that the editor and the loading of definitions be modified to hide the 1K screen limit. The Forth editor should work like a standard text editor. Comments and definitions can be entered anywhere in the file. To maintain compatibility with block-oriented systems such as polyFORTH, each 1K block could be treated like a page. User variables could be defined that would limit the maximum range the editor could use, to avoid overwriting the wrong blocks. In a system that runs under a host operating system, such as F83 under DOS, these "pages" or screen numbers would maintain compatibility with the **VIEW** command. The programmer does not have to keep track of these screen numbers while programming.

Since this is a major change to the current Forth editor philosophy, it may not be acceptable. May I suggest that at the minimum, an editor should be free format, lines terminated with CR-LF and not displayed with line numbers. Even if restrained to the 1K limit, this would be much better than the standard Forth editors. If implemented with a multi-line move command with a large enough buffer to hold a definition when it becomes necessary to move it to the next screen, it would be a great improvement. Then a few words would automate the screen copy or convey, and would move screens as required to make space for new definitions.

If any *FD* reader has a program to read and compile F83 Forth programs from a DOS text file for the IBM-PC, please write to me. I would also like to know if anyone has implemented a Forth editor that uses the IBM-PC cursor keys and produces a straight ASCII text file or the screens as I have outlined above.

Sincerely,

Ramer W. Streed
North Mankato, Minnesota

An invitation to attend the eighth annual

FORML CONFERENCE

The original technical conference
for professional Forth programmers, managers, vendors, and users.

Following Thanksgiving
November 28 - 30, 1986

Asilomar Conference Center

Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California

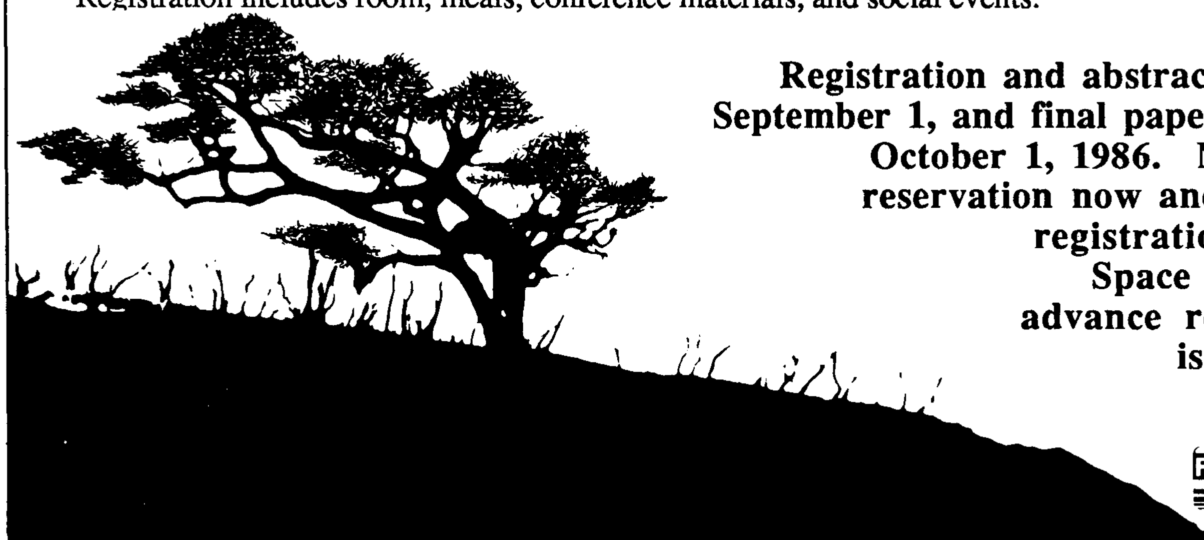
Theme: Extending Forth towards the 87-Standard

FORML isn't part of the Standards Team, but the conference is an opportunity to present your ideas for additions to the Forth standard. Papers are also welcome on other Forth topics. Meet other Forth professionals and learn about the state of the art in Forth applications, techniques, and directions.

To get your registration packet call the FIG Business Office (408) 277-0668 or write to: FORML Registration, Forth Interest Group, P. O. Box 8231, San Jose, CA 95155.

Registration:	$275	Double Room
	$325	Single Room (Limited availability)
	$150	Non-conference guest (Share a double room)

Registration includes room, meals, conference materials, and social events.



Registration and abstracts are due September 1, and final papers are due October 1, 1986. Make your reservation now and get your registration packet. Space is limited, advance registration is required.

FORML

Conditional Teaching

Dear Marlin,

This letter is inspired by Ronald Apra's article on teaching Forth. I've been teaching Forth for several years. My situation is different from Mr. Apra's: I'm teaching university students who already know some other language. Several years ago, however, my wife and I worked with some elementary school students in BASIC, and she has also taught BASIC and Pascal to adults.

Conditional structures, no matter what the language, create a problem for beginners. A student's first language is a barrier. Teaching is often inefficient because we are trying to give students the answers to questions they have never asked. The question in the students' minds is, "Why are we doing this?" My wife suggested to a group of upper elementary students that they write programs to do something *they* would like to do. One girl wanted an "address book." One boy wanted to write a video tennis game (it was suggested that he first try to get a "ball" to bounce back and forth on the screen). Both of these students accomplished their goals. In the process, they had to deal with "How can we do this?" and, as you'd expect, they both came to understand looping and conditional structures quite well. The teaching of computer programming at all levels would benefit by the production of a collection of interesting tasks that develop a need for certain constructs.

"Keep it simple" is indeed a hallmark of Forth. Forth is perhaps the only language whose implementation is simple enough that the language can be learned semantically instead of syntactically. The major unifying principle is "action": each word has an action which it performs when the word is executed. This unifying principle explains why it is desirable that arguments on which the word acts be available before it executes (i.e., reverse Polish syntax). The conditionals can be understood in terms of their action during compilation. A decompiler is an important tool at this stage. If students already know why an **IF ... THEN** construct is needed, and they see how **IF** compiles a conditional branch and **THEN** resolves the displacement, they automatically understand the Forth syntax. fig-FORTH, it should be noted, is advantageous for this approach because system words (like **0BRANCH**) are named and full word names are stored (facilitating decompilation). A student who learns Forth in this way develops a mental image of what each word does — and this eliminates the need to memorize syntax rules.

I repeat again that I'm working with university students with prior programming experience. I'm not sure how far "down" this approach would extend. I am sure, however, that the conceptual simplicity which Forth offers is an important component of its power. I would hate to see "progress" in the direction of words doing a lot of "under the table" manipulation, even if this would provide a superficial simplicity in syntax. I do see potential in

(Letters continued on page 33)

XMODEM Tutorial

John S. James
Santa Cruz, California

The public-domain XMODEM (MODEM7) protocol has come into widespread use for error-free data transfer among personal computers. It has other commercial uses besides, although it is less than ideal for communicating with mainframes over packet-switching networks, a task for which it was not designed. We are publishing this implementation as a tutorial on Forth programming, and also for its practical usefulness. It is not the only XMODEM implementation published in Forth (see references below).

In addition, this two-part article will touch on the philosophy of designing software modules for effective use in libraries. We will also look at compatibility among Forth systems, and at the marketing issues raised by the question of why this author uses the public-domain F83 implementation as a publication language, while using proprietary systems purchased from others for many software-development projects.

Overview

Implementing XMODEM is not as easy as might be assumed. Because of the various states in the execution of the program and their associated timeouts, it may not be obvious at first how to organize the algorithm gracefully in a structured language.

Concerning timeouts, the program should be less strict than the traditional XMODEM when it is used for communicating with mainframes (e.g., services like CompuServe, which supports XMODEM). Since these longer timeout values probably will not interfere with micro-to-micro communication, except to cause some additional delay in recovering from certain errors, we have used them here, with the original XMODEM timeouts in comments.

One challenge for this particular implementation was to make the CRC option efficient in high level. CRC (cyclic redundancy code) was not part of the original XMODEM protocol, but was added later for improved error control. The problem for us is that the CRC algorithm involves much bit shifting, and standard Forth doesn't include efficient shifting. Ordinarily, one would write a code word to do the shifting, but for this example we want to avoid using code in order to run on different computers. So we used some programming tricks, explained below, to implement the CRC efficiently.

This software accumulates the CRC while the characters are coming in, so it must keep up with the characters. But note that the program could be designed to not have to keep up: XMODEM always sends blocks with 128 bytes of data, and reasonable delays between the blocks are allowed. If you have a very slow computer and a very fast modem, you might need to change this program to simply store the characters as they come in, then compute the error check later. The price would be a small but noticeable slowdown in receiving files.

One more challenge in implementing XMODEM is to make it work gracefully with the different variations which exist. Some existing implementations do not talk to some others, mainly due to the timeout differences mentioned above.

What is XMODEM?

XMODEM was developed by Ward Christensen, and has been extended by others. It has been used on bulletin boards since around 1980.

In XMODEM, all data is transmitted in 128-byte blocks. Each block also contains a few bytes of header and error-test information. The receiving program must test each block, and send either an ACK (acknowledge, ASCII 06) to indicate a correct block received, or a NAK (negative acknowledge, ASCII 21 hex) to indicate error. If it sends the NAK, the sending program will retransmit the block.

Each block contains:

- A start-of-header byte, ASCII 01.
- The record number mod 256, starting at 1.
- The ones complement of the record number.
- 128 bytes of data.
- CRC high and low bytes (or a one-byte checksum, if CRC is not used).

The conversation between the computers is receiver driven. This means the sender doesn't do anything until asked by the receiver (except to time out eventually, if the receiver doesn't work properly).

To start the process, both computers must be made ready, one to send and the other to receive. At first, nothing happens. Then the receiver times out, because it failed to get any record; when it times out, it sends a NAK. The NAK tells the sender to retransmit — in this case, to transmit the first record. The receiver ACKs or NAKs it; the transfer process has now started. (If CRC is used, the receiver must send a C the first time — ASCII 43 hex — instead of NAK, in order to tell the sender to use CRC. In either case, the receiver doesn't actually have to wait for the first timeout, but can send the C or the NAK earlier. Some programs try C a few times, and if the sender doesn't respond, they try NAK, in case the sender doesn't support CRC.)

After the file has been transmitted, the sender sends a single character, EOT (end of transmission, ASCII 04). If the receiver gets it, it sends an ACK, and the file transfer is finished.

Unusual Situations

If a NAK gets garbled, the sender won't do anything — until the receiver times out and sends another NAK.

Sometimes an ACK gets garbled. Then the sender will re-transmit a valid record. The receiver, which always tests the record number, throws away the extra copy.

The sender should not treat any non-ACK as a NAK. Especially it should not treat control-S or control-Q, which the receiver may send to ask for a pause in transmission, in this way. If the sender does honor control-S and control-Q (not strictly necessary, since retransmission can take care of most problems), the control-Q to restart transmission may occasionally get lost. Both sides can be designed to handle this case, even if the other side would wait forever, but we haven't done so in this tutorial.

If the record number on a received block is neither the expected one nor a repeat of the previous one, a fatal synchronization error has occurred, and transmission must be terminated. A transmission error in the

```
Scr # 1         A:XMODEM.
\ Compatibility and load screen                              2-17-86JJ

\ These may be needed for other Forth 83 systems, not for F83.
\ : TRUE   -1 ;
\ : FALSE   0 ;
\ : KEY?   ?TERMINAL ;   \ This word is not part of Forth 83 std
\ Define different spelling of NEXT for some assemblers
\ ASSEMBLER DEFINITIONS   : NEXT  NEXT, ;   FORTH DEFINITIONS


\ Load the source code
: XX   13 2 DO  I .  I LOAD  LOOP ;   XX
```

```
Scr # 2         A:XMODEM.
\ Port I/O (IBM-PC-compatible BIOS)                          2-17-86JJ
HEX
CODE CALL-SERIAL    \ x1 n -- x2     Asynchronous DOS call
   DX POP   AX POP   14 INT   AX PUSH   NEXT  END-CODE
: CALL-PORT1    \ x1 -- x2     Call to Port 1
   0 CALL-SERIAL ;
: MINITIALIZE    \ nspeed --     Initialize port
   \ nspeed = 300 or 1200, no parity, 1 stop bit, 8 data bits
   12C ( '300' in hex ) =  IF  43  ELSE  83  THEN
   CALL-PORT1  DROP ;    \ Easily extended to 2400
: MWRITE    \ c --     Write one character to modem
   100 + ( AH=1 )   CALL-PORT1  DROP ;
DECIMAL
```

```
Scr # 3         A:XMODEM.
\ Read a character, using variable timeout                   2-17-86JJ
HEX
: MREAD    \ ntimeout -- c     Read one character from modem
   -1 ( Default, returned if timeout )  SWAP  1 MAX  0 DO
     300 CALL-PORT1 ( Status )  100 AND  IF \ Data ready
       DROP  200 CALL-PORT1 ( Data )  FF AND  LEAVE
     THEN
   9 0  DO  LOOP    \ Delay - adjust for your CPU
   LOOP ;
DECIMAL
```

```
Scr # 4         A:XMODEM.
\ Save all modem I/O, as a record of the session             2-17-86JJ
4000 CONSTANT SAVE-SIZE    \ Size of the terminal save area
CREATE TSAVE  SAVE-SIZE ALLOT  \ Save terminal session for test
VARIABLE SAVE-ADDR   \ Pointer in the save area
: SAVE-INIT   TSAVE SAVE-ADDR !  TSAVE SAVE-SIZE 0 FILL ;
        SAVE-INIT   \ At compile time, for convenient testing
: UMIN     \ u1 u2 -- u      Unsigned minimum
   OVER OVER U<  IF  DROP  ELSE  SWAP DROP  THEN ;
: SAVE     \ c --      Save a character in memory
   SAVE-ADDR @ C!  SAVE-ADDR @ 1+  TSAVE SAVE-SIZE + 1-  UMIN
   SAVE-ADDR ! ;
```

block number will not cause this situation, because the header number and its ones complement must correspond; if they don't, the record will be discarded immediately, before being checked for the fatal error.

Timeouts and retries must keep either computer from hanging indefinitely, no matter what the other does or doesn't do. XMODEM specifies that the timeout between blocks is ten seconds, and within a block the timeout between characters is one second. (We use twenty seconds and ten seconds respectively, as recommended for CompuServe, to assist communication with busy mainframes over packet-switched networks, where delays of over a second are common.) If an error occurs, it should be retried for a total of ten times before the program gives up.

Using the Code

Part I of this two-part article implements the heart of the process. This code lets you receive a record, using the CRC. (The code presented here also includes a dumb-terminal program, named **T**, to let you log on to the remote computer and do whatever is necessary to tell it to start sending a file using XMODEM.)

Before describing how the code works, we'll explain how to use it.

(1) Make sure it compiles. The code runs on the public-domain F83 implementation of the Forth-83 Standard; and screen 1 contains some simple name changes to adapt it to PC/Forth, a Forth-83 Standard implementation from Laboratory Microsystems, Inc. These changes in screen 1 may apply to other Forth-83 Standard systems also.

If your system has a different assembler, the word **CALL-SERIAL** may need to be rewritten; or perhaps your system already has such a word.

(2) If you have a different microprocessor, or have one of the few semi-PC-compatibles which don't support the BIOS calls, you will have to rewrite **MINITIALIZE**, **MREAD** and **MWRITE** for your equipment. (Note that these words actually refer to the port, not the modem.) Your modem too may need some initializing — for example, a dial command and the phone number to dial — but with most modems you can provide these manually by simply typing the characters after you have executed the dumb-terminal program and are talking to the modem. We didn't need to build the modem's initialization into the program, in this example — fortunately, since there are many incompatible modems.

You might want to change the speed from 300 to 1200, in screen 5.

(3) When you get a good compile, execute **T**, the dumb-terminal program. Now you are talking to the modem.

If the system seems to have crashed, it's probably because the modem isn't echoing your characters or saying anything else back to you. You might type a control-Z, which exits from the dumb-terminal program, to make sure you can get back to Forth in order to check that the program hasn't really crashed.

Get back into the terminal mode, and use whatever character or sequence your modem needs to get its attention and into command mode. Usually it's possible to talk to the modem and get something back from it — an "OK," a status report or whatever — before even connecting the telephone. This way you can make sure that everything is working so far, before introducing the added complexity of the remote computer.

(4) Now use the terminal program to talk to a remote computer. Most modems these days are auto-dial — you can type in the phone number as part of a command. In either case, follow the instructions for your particular modem (often easier said than done).

(5) To test the XMODEM record-receive software, call up a bulletin board or a service which supports XMODEM. Do whatever commands are necessary to tell it to start sending a file with XMODEM. Note that for testing this code (in Part I of this article), you need to call a system which can support the CRC error check. (Part II will include code for the simple checksum, also.)

If you use CompuServe or a similar system for the test, make sure to ask for a "text," not "binary" file. Many mainframes have a thirty-six-bit word and normally store text in seven bits per character; so if they must store eight bits, as when saving object programs for micros, they use a special translation such as Intel hex format, which will look like garbage if you receive it.

Once the remote program is in XMODEM and waiting to send data, you have a reasonable timeout period, probably at least a minute, to ask it to start. Get out of the dumb-terminal program and back to Forth with the control-Z. You're still online with the remote computer, and can execute **T** again later to get back into the terminal mode.

To start the remote system, send the letter C (ASCII 67 decimal). Since the data may start coming quickly, the command to receive it should be on the same line, as in

67 **MWRITE XGET**

XGET is a short test word defined on screen 12. It prints the number of characters read (should be 133), and a flag (0 = good, 1 = end-of-file, 2 = an error such as too few characters or bad CRC).

The record transmission will take a little over one second at 1200 bps, four seconds at 300. If everything worked, **XGET** will print 133 and zero. Dump some bytes from **DATA** to see if you got the start of header (01), the record number (01), its ones complement (FE hex) and the data. Or just use **DATA 3 + 128 TYPE** to see the text. (Some Forth systems will **TYPE** carriage returns and line feeds as control characters, others will perform them instead.)

If you got 133 bytes but an error, it is possible that the software is working properly and XMODEM detected a transmission error, as it should. If it seemed that nothing happened, wait half a minute or so for any possible timeout, and then check the results from **XGET**. If one or two characters are missing, there may have been transmission errors. If no characters were received, perhaps the remote system does not support CRC; try

<NAK> **XWRITE XGET**

and see if 132 characters are received; there will be timeout and an error because the code given here does not support the simple checksum, only the CRC.

To try for another record, use

ACK XGET to get the next one, or

NAK XGET for retransmission of the last record received.

There is another way to check what has happened. The dumb-terminal program keeps a record of your session — all characters sent both ways — at **TSAVE**. If things don't work, you can dump that data.

(6) To end the test, it's helpful to get back to the terminal mode and log off the remote system. But if you haven't reached the end of file, it may be hard to get the sending program to abort the transmission. If nothing else will do, unplug the phone.

How the Code Works

Screens 2 and 3 provide the low-level I/O — **MINITIALIZE**, **MREAD** and **MWRITE** — to initialize, and to read and write characters. As explained above, you may need to rewrite these words.

Note that **MREAD** accepts a timeout value in milliseconds. You may want to adjust the delay loop in screen 4 to make the values relatively accurate for your computer. There are better ways to handle timing, of course, but we kept it simple for this tutorial.

Screen 4 defines words to save a record of the session — a copy of all I/O — in a memory area which can be examined later to trace problems.

Screen 4 illustrates that comparisons of addresses must be unsigned — using the Forth-83 Standard word **U<**. Otherwise, the result would be correct only if both addresses were on the same side of the 32K boundary (in a 64K system). A program could work perfectly until a recompilation just happened to cause a buffer to span that boundary. So instead of using **MIN** to get a minimum, as part of the process of preventing the trace of the session from overflowing its save area, we defined **UMIN**, an unsigned minimum, and used it instead.

Screen 5 implements a dumb-terminal program, **T**. We use a short name to make it easy to get into and out of terminal mode (control-Z gets out of the terminal mode and back to Forth). **T** includes the machinery to save the record of a session — all characters sent either way. Note that **SAVE-INIT** is executed at compile time; otherwise it would be easy to forget when testing. In a product, **T** would execute **SAVE-INIT**; but here that wouldn't be appropriate, because during testing you should be able to get into and out of terminal mode without wiping out the record of the session so far.

Screen 5 also redefines **MREAD** and **MWRITE**, to create new versions which save a copy of the data they transfer. Note that we could not use these redefinitions in **T**, because **T** is constantly doing **MREAD** which times out; it would fill the session-save area with garbage.

Screen 6 sets up constants and variables for the XMODEM code itself.

Screen 7 does the "line purge" — waiting until the sender stops transmitting — which XMODEM requires before a NAK, to avoid possible confusion.

Screens 8 and 9 are the hard part — computing the CRC, and doing so efficiently in high level.

Here is the algorithm. Start with a 16-bit accumulator set to zero. Take the 128 data

```
Scr # 5          A:XMODEM.
\ Dumb terminal, for test. 'T' to run, control-Z exit  2-17-86JJ
: T    \ --       Dumb terminal program
  300 MINITIALIZE    \ 300, 1200, or other
  FALSE \ Loop control - set to TRUE to exit
  BEGIN
    0 MREAD   \ No need for timeout delay here
    DUP -1 =  IF  DROP  ELSE  127 AND  DUP SAVE  EMIT  THEN
    KEY? IF   \ If key typed, send it, unless control-Z
      KEY  DUP 26 =  IF  DROP DROP 1  \ Exit
      ELSE  DUP SAVE  MWRITE  THEN
    THEN
  DUP UNTIL  DROP ;

\ Apply SAVE to later I/O - can't use this in T, due to timeout
: MREAD   MREAD  DUP SAVE ;
: MWRITE   DUP SAVE  MWRITE ;
```

```
Scr # 6          A:XMODEM.
\ XMODEM variables and constants                       2-17-86JJ
VARIABLE ARECORD    \ Address of 133-byte area
VARIABLE NCHAR      \ Number of characters transmitted
VARIABLE XTIMEOUT    \ Timeout value for read, in milliseconds
VARIABLE DATA-BYTE   \ Used in CRC
 1 CONSTANT <SOH>
 4 CONSTANT <EOT>
 6 CONSTANT <ACK>
21 CONSTANT <NAK>
10000 CONSTANT CTIMEOUT   \ 10 sec, timeout between characters
20000 CONSTANT BTIMEOUT   \ 20 sec, timeout between blocks
\ In traditional XMODEM, above timeouts are 1 sec and 10 sec
 1000 CONSTANT PTIMEOUT   \ Try 1 second to purge the line
```

```
Scr # 7          A:XMODEM.
\ Purge the line (wait for transmission to stop)       2-17-86JJ
: XPURGE    \ --       Wait till transmission stops
  BEGIN
    PTIMEOUT MREAD   \ Wait till no more characters
  -1 =  UNTIL ;
```

```
Scr # 8          A:XMODEM.
\ Compute CRC                                          2-17-86JJ
HEX
: 1BIT    \ xaccum1 mask -- xaccum2     Apply 1 bit of DATA-BYTE
  SWAP ( Save mask )   DUP 8000 AND  IF   \ Bit to shift out
    DUP + \ Shift left
    SWAP DATA-BYTE @ AND ( Apply mask )   IF 1+ THEN \ "Shift"
    1021 XOR    \ CRC polynomial
  ELSE   \ Same thing, only don't do the CRC
    DUP +
    SWAP DATA-BYTE @ AND   IF  1+  THEN
  THEN ;

DECIMAL
```

bytes, and shift each bit (high bit first) of each byte into the accumulator. Throw away anything shifted out of the accumulator — but whenever a 1 bit is shifted out, exclusive-or the accumulator with the magic number 1021 hex (a number specified in the CCITT CRC-16 standard).

We will do this with the 128 data bytes as they come in. The algorithm also requires two zero bytes to be put through at the end. The final value in the accumulator should match the CRC bytes which were transmitted.

How do we do all this efficiently without an efficient shift — except a left shift by adding a number to itself, which doesn't even keep track of anything shifted out?

This program keeps the accumulator on the stack. **UPDATE-CRC**, on screen 9, takes advantage of the fact that the data byte need not be physically shifted, if eight different masks are applied to see if there is a bit in the position from which it would have been shifted out. If so, a bit is simply added to the accumulator (after the accumulator has been "shifted" by adding to it a copy of itself). The **1+** in **1BIT** causes the effect of shifting a 1 into the accumulator.

To get around the fact that **+** doesn't keep track of overflow, simply test the high bit of the accumulator with **8000 AND** (hex) before adding to itself.

The result of this test needs to be used later — after the rest of the work on the accumulator. To save a little time, **1BIT** doesn't store that result, but instead has parallel code sequences — in effect, "storing" the bit shifted out in the control of the program itself.

In screen 9, **GET-CRC** puts only the 128 data bytes (of the 133 bytes transmitted) and the two required zeros into the CRC. (We don't actually use **GET-CRC** here, since we accumulate the CRC while the characters are being received; but the word can be handy for testing.)

Note that **CRC-BAD?** in screen 9 picks up the transmitted CRC byte-by-byte, not with **@**, to avoid byte-order dependency which would cause the program to fail on some CPUs. The rule is that **@** should not pick up what **!** did not put down; **C@** is always okay, however. Forth programmers seldom have to worry about byte order — unless the data has come in from another computer, as here.

The rest of the code is fairly straightforward. **XREAD** reads n characters, or until timeout. **XCHECK** tests the data received. **XGET**, **ACK** and **NAK** are convenient test words to run from the terminal.

Notes on the Code

The code example will not work with some external modems due to RS-232 incompatibilities. **MREAD** and **MWRITE** will timeout in one second and not transfer any data. You may be able to avoid the problem by setting switches on the modem, or by using a special cable.

Or you could patch **MREAD** and **MWRITE** to bypass the operating system and access the port directly. Replace the definition of **MWRITE** with:

3F8 PC!

where **PC!** writes one byte to a port. In **MREAD**, replace **200 CALL-PORT1** with **3F8 PC@**. (Some systems may need one of these patches without the other, or may need a more elaborate **MREAD**.)

Also note that the crude dumb terminal program **T** may lose characters between lines, while the screen is scrolling. XMODEM is not affected, however; the dumb terminal is used only to navigate through the remote system to get its XMODEM started. We chose not to complicate the article with the buffered version of a word used only for testing.

Discussion — Forth Libraries

I am implementing a Forth library system to allow developers to use off-the-shelf modules (see "Forth Component Libraries," reference below). The code here does not use the library facilities, since they are not yet available; but this code was designed to be adapted and used in the library.

The library will hide the internals. It will show the XMODEM words to the outside world on three different levels. The low level will be **MINITIALIZE**, **MREAD** and **MWRITE** — not really XMODEM at all. The middle level will have words like **XGET** which read and write records. The highest level will receive and send whole files, creating, opening and closing them, etc.

Note:

(1) The low-level words don't take a full set of all possible transmission options. For example, **MINITIALIZE** takes advantage of the fact that XMODEM specifies no parity, eight data bits and one stop bit; usually these values will work for other purposes

```
Scr # 9          A:XMODEM.
 \ Compute CRC                                         2-17-86JJ
HEX
: UPDATE-CRC   \ xaccum1 byte -- xaccum2
   DATA-BYTE !
   80 1BIT   40 1BIT   20 1BIT   10 1BIT   \ Apply each bit
   08 1BIT   04 1BIT   02 1BIT   01 1BIT ;
DECIMAL
: GET-CRC    \ arecord -- xcrc     Test only - compute it
   0    \ Start CRC accumulator
   SWAP 3 +  128  OVER + SWAP  DO  I C@ UPDATE-CRC  LOOP
   0 UPDATE-CRC   0 UPDATE-CRC ;
: CRC-BAD?   \ xcrc -- f      Test for bad CRC
   ARECORD @ 131 + C@ 256 *   ARECORD @ 132 + C@   +   <> ;
   \ Don't just fetch the CRC, because of byte-order dependency
```

```
Scr # 10         A:XMODEM.
\ Test for garbled data                                 2-17-86JJ
: GARBLED?   \ xcrc ftimeout -- fbad     Test for bad record
   \ Note: doesn't check for unexpected block #;
   \ do that at higher level.
   TRUE =    \ Timed out
   ARECORD @ C@ <SOH> <>  OR    \ No <SOH>
   ARECORD @ 1+ C@  ARECORD @ 2+ C@   +  255 <>  OR \ Bad compl
   SWAP CRC-BAD?  OR ;
```

```
Scr # 11         A:XMODEM.
\ Read n characters (or less if timeout)                2-17-86JJ
: XREAD   \ nexpected -- xcrc nread     Return CRC, # read
   ARECORD @ 133 0 FILL    \ Avoid confusion with past data
   BTIMEOUT XTIMEOUT !   \ Use block timeout on first time thru
   0 NCHAR !   0 SWAP   \ Initial CRC accumulator
   ( nexpected ) 0 DO    \ Read the characters
     XTIMEOUT @ MREAD   CTIMEOUT XTIMEOUT !    \ Char. timeout
     DUP -1 >  IF   \ Did not time out
       DUP ARECORD @ I + C!    \ Store the character
       I 3 <  I 130 >  OR  IF  DROP  ELSE  UPDATE-CRC  THEN
         \ Put only the data bytes into the CRC
       I 1+ NCHAR !     \ Update count of characters
     ELSE  ( timed out )  DROP LEAVE  THEN
   LOOP  0 UPDATE-CRC  0 UPDATE-CRC  NCHAR @ ;
```

```
Scr # 12         A:XMODEM.
\ Check the received record                             2-17-86JJ
: XCHECK    \ xcrc nread -- nflag     Flag 0=good, 1=eof, 2=err
    DUP 1 =  ARECORD @ C@ <EOT> =  AND  IF  \ End of file
      DROP DROP  1  \ Return result
    ELSE  \ Regular record
      OVER OVER 133 <  ( timeout? )  GARBLED?  IF   \ Bad record
        DROP DROP  2  \ Return result
      ELSE  \ Good record
        DROP DROP  0  \ Result
    THEN  THEN ;

\ Handy words for testing
CREATE DATA  133 ALLOT   DATA ARECORD !    \ Set up I/O area
: XGET    ( -- )   133 XREAD  DUP .  XCHECK . ; \ Print n, flag
: ACK    <ACK> MWRITE ;      : NAK   XPURGE  <NAK> MWRITE ;
\ To start, use   67 MWRITE   (decimal)
```

also. We want to make the words easy to use for getting in quickly and doing a job. But what happens when other options are needed? The central question is how to expand the software library in a graceful, upwardly compatible way.

One approach is to define another word, loaded with stack arguments for all the options, and then redefine **MINITIALIZE** in terms of it. Another approach is to define an argument-communication area for **MINITIALIZE**, and let it default to values which cause **MINITIALIZE** to behave the same unless the programmer does something with the new word. Either way allows upwardly compatible inclusion of new options, even ones not thought of in advance.

(2) It's okay to use temporary storage areas (variables, here). The library uses a re-entrant system instead of ordinary variables, but variables are all right for now.

(3) How should routines handle errors? They must not print error messages, because they may be used when no terminal or at least no person is available. So they must signal whatever called them, so that the higher-level, application-oriented logic can decide on appropriate action.

Routines can pass an error flag to the stack. But exhaustive error reporting would make routines too cumbersome for quick and easy use. For example, **MINITIALIZE** could make all kinds of reports, depending on what status information was returned by the particular operating system or port.

So we took the easy approach and had **MINITIALIZE** throw the status and error information away. Later, it could be generalized with a new word as explained above, or by use of an error-communication area, just a place where routines dump whatever they want to report, and don't change otherwise. Higher-level words which call the routine can, if they care, first clear the error area to null (impossible) values, then check it later to see what happened. But the user of **MINITIALIZE**, who doesn't need to know about any of this, still has an easy and convenient word for ordinary uses.

Some errors fit naturally at higher levels, and should not be handled by low-level routines. For example, the word **XCHECK** does test whether a block number and its ones complement correspond, but it should not test whether the block number is the expected one. If it did, it would need to keep track of what record the application program was expecting and would have more complicated arguments.

Discussion — Forth Marketing

Screen 1 includes code to compile this software on the PC/Forth system mentioned above. In fact, I developed this XMODEM implementation to run on this LMI Forth, choosing it among many competent systems because it had target compilation and other support for the particular chip I was using — support which would have taken weeks to write from scratch.

This example illustrates an important aspect of the sometimes-troubled relationship between public-domain and proprietary Forth systems. Publication and software development often have different requirements.

For publication, while work in any system is welcome, it can be helpful to standardize on a publication language when possible. A large coding example will need a few words outside of the Forth standard, as we saw with **KEY?** vs. **?TERMINAL** here; and some published papers will want to deal with internals. It would be helpful to have a common base to talk from.

It wouldn't be fair to standardize on the system of one vendor, excluding all the others. Nor would it be wise to give one company such control over public discourse. Also, the publication language should have source code available and permit publication of parts of that code for teaching examples or in order to discuss modifications. Even if permission may be available it presents practical problems, such as the fact that you may not know specifically what to ask for until having already done the work, and then you do not know how long it would take to get a response. Using a public-domain publication language avoids these problems.

But vendors have lost sales due to the availability of free Forth systems. If they get hurt, much of the future development of Forth won't get done. What seems to be happening is that we have passed the time when a Forth system by itself, with yet another competent editor, assembler and even target compiler, is the heart of a product. What counts more is the support of particular application-oriented tasks that certain users want to do. Potential users often complain about having to write from scratch, in Forth, tools which could have been available off the shelf if they used C. The need is there.

It is difficult to define these markets, however. When users are asked what they would want in Forth, they request vastly different and often incompatible application routines, tools or improvements. It seems that the future is in discovering application-oriented niche markets for which particular vendors can provide solid, effective support.

References

1. Christensen, Ward. "Modem Protocol Overview," January 1982, available on various bulletin boards.

2. Ham, Michael and Michael McNeil, Stephen Martin, William Lindow. "An Industry Look at Fifteen Forths," *Computer Language*, Volume 2, Number 8, August 1985.

3. James, John S. "Forth Component Libraries," *Forth Dimensions*, Volume 7, Number 4, November/December 1985.

4. Krantz, Donald. "Christensen Protocols in C," *Dr. Dobb's Journal*, Volume 10, Issue 6, June 1985.

5. Taylor, Robert. "SEND and RCV: A Forth Implementation of the XMODEM Protocol," *Dr. Dobb's Journal*, Volume 8, Issue 9, September 1983.

6. Author unknown. "XMODEM and CompuServe," October 1984, available on CompuServe.

FORTH INTEREST GROUP MAIL ORDER FORM

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

MEMBERSHIP
IN THE FORTH INTEREST GROUP

108 - MEMBERSHIP in the FORTH INTEREST GROUP & Volume 8 of FORTH DIMENSIONS. No sales tax, handling fee or discount on membership. See the back page of this order form.

The Forth Interest Group is a worldwide non-profit member-supported organization with over 4,000 members and 90 chapters. FIG membership includes a subscription to the bi-monthly publication, FORTH Dimensions. FIG also offers its members publication discounts, group health and life insurance, an on-line data base, a large selection of Forth literature, and many other services. Cost is $30.00 per year for USA, Canada & Mexico; all other countries may select surface ($37.00) or air ($43.00) delivery.

The annual membership dues are based on the membership year, which runs from May 1 to April 30.

When you join, you will receive issues that have already been circulated for the current volume of Forth Dimensions and subsequent issues will be mailed to you as they are published.

You will also receive a membership card and number which entitles you to a 10% discount on publications from FIG. Your member number will be required to receive the discount, so keep it handy.

HOW TO USE THIS FORM

1. Each item you wish to order lists three different Price categories:

   Column 1 - USA, Canada, Mexico
   Column 2 - Foreign Surface Mail
   Column 3 - Foreign Air Mail

2. Select the item and note your price in the space provided.
3. After completing your selections enter your order on the fourth page of this form.
4. Detach the form and return it with your payment to the **Forth Interest Group**.

FORTH DIMENSIONS BACK VOLUMES

The six issues of the volume year (May — April)

101 - Volume 1 FORTH Dimensions (1979/80)$15/16/18 ______
102 - Volume 2 FORTH Dimensions (1980/81)$15/16/18 ______
103 - Volume 3 FORTH Dimensions (1981/82)$15/16/18 ______
104 - Volume 4 FORTH Dimensions (1982/83)$15/16/18 ______
105 - Volume 6 FORTH Dimensions (1983/84)$15/16/18 ______
106 - Volume 5 FORTH Dimensions (1984/85)$15/16/18 ______
107 - Volume 7 FORTH Dimensions (1985/86)$20/21/24 ______

FORML CONFERENCE PROCEEDINGS

FORML PROCEEDINGS — FORML (the Forth Modification Laboratory) is an informal forum for sharing and discussing new or unproven proposals intended to benefit Forth. Proceedings are a compilation of papers and abstracts presented at the annual conference. FORML is part of the Forth Interest Group.

310 - FORML PROCEEDINGS 1980 . . . . $30/33/40 ________
Technical papers on the Forth language and extensions.

311 - FORML PROCEEDINGS 1981 (2V) $45/48/55 ________
Nucleus layer, interactive layer, extensible layer, metacompilation, system development, file systems, other languages, other operating systems, applications and abstracts without papers.

312 - FORML PROCEEDINGS 1982 . . . . $30/33/40 ________
Forth machine topics, implementation topics, vectored execution, system development, file systems and languages, applications.

313 - FORML PROCEEDINGS 1983 . . . . $30/33/40 ________
Forth in hardware, Forth implementations, future strategy, programming techniques, arithmetic & floating point, file systems, coding conventions, functional programming applications.

314 - FORML PROCEEDINGS 1984 . . . . $30/33/40 ________
Expert systems in Forth, using Forth, philosophy, implementing Forth systems, new directions for Forth, interfacing Forth to operating systems, Forth systems techniques, adding local variables to Forth.

315 - FORML PROCEEDINGS 1985 . . . . $35/38/45 ________
Also includes papers from the 1985 euroFORML Conference. Applications: expert systems, data collection, networks. Languages: LISP, LOGO, Prolog, BNF. Style: coding conventions, phrasing. Software Tools: decompilers, structure charts. Forth internals: Forth computers, floating point, interrupts, mulitasking, error handling.

BOOKS ABOUT FORTH

200 – ALL ABOUT FORTH .... $25/26/35 ________
Glen B. Haydon
An annotated glossary for MVP Forth; a 79-Standard Forth.

216 – DESIGNING & PROGRAMMING PERSONAL EXPERT SYSTEMS .. $19/20/29 ________
Carl Townsend & Dennis Feucht
Introductory explanation of AI-Expert System Concepts. Create your own expert system in Forth. Written in 83-Standard.

217 – F83 SOURCE .... $25/26/35 ________
Dr. C. H. Ting
A complete listing of F83 including source and shadow screens. Includes introduction on getting started.

218 – FOOTSTEPS IN AN EMPTY VALLEY (NC4000 Single Chip Forth Engine) $25/26/35 ________
Dr. C. H. Ting
A thorough examination and explanation of the NC4000 Forth chip including the complete source to cmForth from Charles Moore.

219 – FORTH: A TEXT AND REFERENCE $25/26/35 ________
Mahlon G. Kelly & Nicholas Spies
A text book approach to Forth with comprehensive references to MMS Forth and the 79 and 83 Forth Standards.

220 – FORTH ENCYCLOPEDIA .... $25/26/35 ________
Mitch Derick & Linda Baker
A detailed look at each fig-Forth instruction.

225 – FORTH FUNDAMENTALS, V.1 ... $16/17/20 ________
Kevin McCabe
A textbook approach to 79-Standard Forth

230 – FORTH FUNDAMENTALS, V.2 ... $13/14/18 ________
Kevin McCabe
A glossary.

232 – FORTH NOTEBOOK .... $25/26/35 ________
Dr. C. H. Ting
Good examples and applications. Great learning aid. PolyFORTH is the dialect used. Some conversion advice is included. Code is well documented.

233 – FORTH TOOLS .... $22/23/32 ________
Gary Feierbach & Paul Thomas
The standard tools required to create and debug Forth-based applications.

235 – INSIDE F-83 .... $25/26/35 ________
Dr. C. H. Ting
Invaluable for those using F-83.

237 – LEARNING FORTH .... $17/18/27 ________
Margaret A. Armstrong
Interactive text, introduction to the basic concepts of Forth. Includes section on how to teach children Forth.

240 – MASTERING FORTH .... $18/19/22 ________
Anita Anderson & Martin Tracy
A step-by-step tutorial including each of the commands of the Forth-83 International Standard; with utilities, extensions and numerous examples.

245 – STARTING FORTH (soft cover) ... $22/23/32 ________
Leo Brodie
A lively and highly readable intruduction with exercises.

246 – STARTING FORTH (hard cover) .. $24/25/29 ________
Leo Brodie

255 – THINKING FORTH (soft cover) $16/17/20
Leo Brodie
The sequel to "Starting Forth". An intermediate text on style and form.

265 – THREADED INTERPRETIVE LANGUAGES ... $25/26/35
R. G. Loelinger ________
Step-by-step development of a non-standard Z-80 Forth.

270 – UNDERSTANDING FORTH .... $3.50/5/6 ________
Joseph Reymann
A brief introduction to Forth and overview of its structure.

ROCHESTER PROCEEDINGS

The Institute for Applied Forth Research, Inc. is a non-profit organization which supports and promotes the application of Forth. It sponsors the annual Rochester Forth Conference.

321 – ROCHESTER 1981
(Standards Conference) .... $25/28/35 ________
79-Standard, implementing Forth, data structures, vocabularies, applications and working group reports.

322 – ROCHESTER 1982
(Data bases & Process Control) ... $25/28/35 ________
Machine independence, project management, data structures, mathematics and working group reports.

323 – ROCHESTER 1983
(Forth Applications) .... $25/28/35 ________
Forth in robotics, graphics, high-speed data acquisition, real-time problems, file management, Forth-like languages, new techniques for implementing Forth and working group reports.

324 – ROCHESTER 1984
(Forth Applications) .... $25/28/35 ________
Forth in image analysis, operating systems, Forth chips, functional programming, real-time applications, cross-compilation, multi-tasking, new techniques and working group reports.

325 – ROCHESTER 1985
(Software Management & Engineering) $20/21/24 _____
Improving software productivity, using Forth in a space shuttle experiment, automation of an airport, development of MAGIC/L, and a Forth-based business applications language; includes working group reports.

THE JOURNAL OF FORTH APPLICATION & RESEARCH

A refereed technical journal published by the Institute for Applied Forth Research, Inc.

401 – JOURNAL OF FORTH RESEARCH V.1 #1
Robotics. .... $15/16/18 ________

402 – JOURNAL OF FORTH RESEARCH V.1 #2
Data Structures. .... $15/16/18 ________

403 – JOURNAL OF FORTH RESEARCH V.2 #1
Forth Machines. .... $15/16/18 ________

404 – JOURNAL OF FORTH RESEARCH V.2 #2
Real-Time Systems. .... $15/16/18 ________

405 – JOURNAL OF FORTH RESEARCH V.2 #3
Enhancing Forth. .... $15/16/18 ________

406 – JOURNAL OF FORTH RESEARCH V.2 #4
Extended Addressing. .... $15/16/18 ________

407 – JOURNAL OF FORTH RESEARCH V.3 #1
Forth-based laboratory systems and data structures.
.... $15/16/18 ________

REPRINTS

420 – BYTE REPRINTS .... $5/6/7 ________
Eleven Forth articles and letters to the editor that have appeared in *Byte* Magazine.

DR. DOBB'S JOURNAL

This magazine produces an annual special Forth issue which includes source-code listing for various Forth applications.

422 - DR. DOBB'S 9/82 . . . . $5/6/7 ________
423 - DR. DOBB'S 9/83 . . . . $5/6/7 ________
424 - DR. DOBB'S 9/84 . . . . $5/6/7 ________
425 - DR. DOBB'S 10/85 . . . . $5/6/7 ________
426 - DR. DOBB'S 7/86 . . . . $5/6/7 ________

HISTORICAL DOCUMENTS

501 - KITT PEAK PRIMER . . . . $25/27/35 ________
One of the first institutional books on Forth. Of historical interest.

502 - Fig-FORTH INSTALLATION MANUAL $15/16/18 ________
Glossary model editor — We recommend you purchase this manual when purchasing the source-code listing.

503 - USING FORTH . . . . $20/21/23 ________
FORTH, Inc.

REFERENCE

305 - FORTH 83-STANDARD . . . . $15/16/18 ________
The autoritative description of 83-Standard Forth. For reference, not instruction.

300 - FORTH 79-STANDARD . . . . $15/16/18 ________
The authoritative description of 79-Standard Forth. Of historical interest.

316 - BIBLIOGRAPHY OF FORTH REFERENCES
2nd edition, Sept. 1984 . . . . $15/16/18 ________
An excellent source of references to articles about Forth throughout microcomputer literature. Over 1300 references.

ASSEMBLY LANGUAGE SOURCE CODE LISTINGS

Assembly Language Source Listings of fig-Forth for Specific CPUs and machines with compiler security and variable length names.

514 - 6502/SEPT 80 . . . . $15/16/18 ________
515 - 6800/MAY 79 . . . . $15/16/18 ________
516 - 6809/JUNE 80 . . . . $15/16/18 ________
517 - 8080/SEPT 79 . . . . $15/16/18 ________
518 - 8086/88/MARCH 81 . . . . $15/16/18 ________
519 - 9900/MARCH 81 . . . . $15/16/18 ________
521 - APPLE II/AUG 81 . . . . $15/16/18 ________
523 - IBM-PC/MARCH 84 . . . . $15/16/18 ________
526 - PDP-11/JAN 80 . . . . $15/16/18 ________
527 - VAX/OCT 82 . . . . $15/16/18 ________
528 - Z80/SEPT 82 . . . . $15/16/18 ________

MISCELLANEOUS

601 - T-SHIRT SIZE ________________
Small, Medium, Large and Extra-Large.
White design on a dark blue shirt. . $10/11/12 ________

602 - POSTER (BYTE Cover) . . . . $5/6/7 ________
616 - HANDY REFERENCE CARD . . . . FREE ________
683 - FORTH-83 HANDY REFERENCE CARD . . FREE _____

FORTH MODEL LIBRARY

The model applications disks described below are new additions to the Forth Interest Group's library. These disks are the first releases of new professionally developed Forth applications disks. Prepared on 5 1/4" disks, they are IBM MSDOS 2.0 and up compatible. The disks are compatible with Forth-83 systems currently available from several Forth vendors. Macintosh 3 1/2" disks are available for MasterFORTH systems only.

Forth-83 Compatibility IBM MSDOS

Laxen/Perry F83	LMI PC/FORTH 3.0
MasterFORTH 1.0	TaskFORTH 1.0
PolyFORTH® II	

Forth-83 Compatibility Macintosh

MasterFORTH

ORDERING INFORMATION

701 - A FORTH LIST HANDLER V.1 . . . . $40/43/45 ________
by Martin J. Tracy
Forth is extended with list primitives to provide a flexible high-speed environment for artificial intelligence. ELISA and Winston & Horn's micro-LISP are included as examples. Documentation is included on the disk.

702 - A FORTH SPREADSHEET V.2 . . . . $40/43/45 ________
by Craig A. Lindley
This model spreadsheet first appeared in Forth Dimensions Volume 7, Issue 1 and 2. These issues contain the documentation for this disk.

703 - AUTOMATIC STRUCTURE CHARTS V.3 $40/43/45_____
by Kim R. Harris
These tools for the analysis of large Forth programs were first presented at the 1985 FORML conference. Program documentation is contained in the 1985 FORML Proceedings.

Please specify disk size when ordering . . . . ________

FORTH INTEREST GROUP

P.O. BOX 8231 SAN JOSE, CALIFORNIA 95155 408/277-0668

Name ______________________

Member Number ______________________

Company ______________________

Address ______________________

City ______________________

State/Prov. ______________ ZIP ______________

Country ______________________

Phone ______________________

OFFICE USE ONLY

By ________ Date ________ Type ________

Shipped By ________ Date ________

UPS Wt. ________ Date ________

USPS Wt. ________ Amt. ________

BO Date ________ By ________

Wt. ________ Amt. ________

ITEM #	TITLE	AUTHOR	QTY	UNIT PRICE	TOTAL
108	MEMBERSHIP →				SEE BELOW

SUBTOTAL	
10% MEMBER DISCOUNT MEMBER #	
SUBTOTAL	
CA. RESIDENTS SALES TAX	
HANDLING FEE	$2.00
MEMBERSHIP FEE ☐ NEW ☐ RENEWAL $30/37/43	
TOTAL	

☐ Check enclosed (payable to: **FORTH INTEREST GROUP**)

☐ VISA ☐ MASTERCARD

Card # ______________________

Expiration Date ______________________

Signature ______________________

($15.00 minimum on charge orders)

PAYMENT MUST ACCOMPANY ALL ORDERS

MAIL ORDERS
Send to:
Forth Interest Group
P.O. Box 8231
San Jose, CA 95155

PHONE ORDERS
Call 408/277-0668 to place credit card orders or for customer service. Hours: Monday-Friday, 9am-5pm PST.

PRICES
All orders must be prepaid. Prices are subject to change without notice. Credit card orders will be sent and billed at current prices. $15 minimum on charge orders. Checks must be in US$, drawn on a US Bank. A $10 charge will be added for returned checks.

POSTAGE & HANDLING
Prices include shipping. A $2.00 handling fee is required with all orders.

SHIPPING TIME
Books in stock are shipped within five days of receipt of the order. Please allow 4-6 weeks for out-of-stock books (delivery in most cases will be much sooner).

SALES TAX
Deliveries to Alameda, Contra Costa, San Mateo, Los Angeles, Santa Cruz and San Francisco Counties, add 6½%. Santa Clara County, add 7%; other California counties, add 6%.

8-1-86

On-Line Documentation

79

John J. Wavrik
Solana Beach, California

The best form of documentation for a Forth application is the source code. Some Forth systems provide a word **LOCATE** (or **VIEW**) so that the phrase **LOCATE** *word* will show the screen on which *word* has been defined. It is implemented by extending the headers of dictionary entries to include the block number on which the word was defined (and, on some systems, a file identifier).

The **LOCATE** word itself can be defined in a system-independent way. It is assumed that each header has been equipped with a "screen field" in addition to the usual name, link, code and parameter fields. SFA will denote the address of the screen field for the entry at hand.

Please refer to Figure One. The constant **WALL** marks the boundary between the words which are compiled prior to **LOCATE** (those which may not have the additional field) and those compiled subsequently. Several system-dependent words must be loaded before this:

KB? (sfa -- t=defined from keyboard)

The stored block number for a keyboard entry will be either 0 or -1 on most systems.

CFA>SFA (cfa -- sfa)

SHOW-SCREEN (sfa --)

This word will typically just list the screen. The user may choose to also make the screen available for editing.

SAVE-ENVIRONMENT

RESTORE-ENVIRONMENT

The **LOCATE** word will use a block buffer and change the contents of one or more system variables. If desired, **LOCATE** can be made to function transparently and restore the contents of the buffers, etc.

LOCATE is a word best supplied with the Forth system. Its *ex post facto* installation requires modification of the system word(s) responsible for making headers, and so a knowledge of the (system-dependent) process by which this takes place. Some examples will clarify the nature of the task. fig-FORTH mandates a word **CREATE** which produces headers. More recent Forth standards (Forth-79 and Forth-83) regard the manner in which headers are produced (and even the configuration of the header) to be implementation dependent. Information about the making of headers must be obtained by examination of the source code for the system, if it is available, or by decompilation. The standard defining words **:**, **CODE**, **CONSTANT** and **VARIABLE** should be decompiled to identify the word responsible for making the header. In some systems, this word is an orphan (i.e., a word without a name). It will have to be decompiled using its absolute code address.

fig-FORTH

fig-FORTH is a public-domain version of Forth distributed by the Forth Interest Group starting in 1979. Its wide availability for a variety of processors made it a *de facto* standard.

fig-FORTH uses the word **CREATE** to produce new headers. (The behavior of this word is different than that in the Forth-79 and Forth-83 Standards.) The definition of **CREATE** begins:

```
: CREATE  -FIND  IF DROP ...
```

The installation of **LOCATE** is shown in Figure Two. When reading from the keyboard, **BLK** contains zero. We do not restore the environment after **LOCATE**.

Forth-79 Systems:
MMS-FORTH v. 2.4

In this system from Miller Microcomputer Services **HEAD**, is the word responsible for making headers (this word was an orphan in MMS-FORTH v. 2.0). See Figure Three-b. For this system, all name fields are four bytes. The new field is eight bytes before the code field. When the system reads from the keyboard, **BLK** contains zero. Here we completely restore the environment after **LOCATE** (Figure Three-c).

In this system, data about the two buffers (numbered zero and one) is stored in an array **BUFFDATA**. The first two bytes of *n* **BUFFDATA** contain the number of the block currently in buffer *n*. The next byte indicates the order of referencing (two indicates the next buffer to be loaded). The remaining byte is the update byte (one if the block is marked, zero otherwise). **SCR** is a variable containing the number of the last screen listed or edited. All of this information is saved. To return blocks to the original buffers, we first identify the buffer containing the screen just listed by **LOCATE**. We force the next block to be loaded in this buffer. We reload the block originally in this buffer. We restore all the **BUFFDATA** information.

This example indicates what could be required to restore the total environment. It is more common to restore the environment only partially. If, in the midst of editing, **LOCATE** is used to check a definition, it is possible to ensure that the current editing block is restored to memory (but not in the original buffer and without restoring the update flag), as in Figure Three-d.

Forth-79 Systems:
MVP-FORTH

The Forth available from Mountain View Press uses **CREATE** to make headers (Figures Four-a and Four-b). This system stores full names, as in fig-FORTH, and it uses the fig-FORTH nomenclature for accessing the fields in a header. When the system reads from the keyboard, **BLK** contains zero. We will make no attempt to restore the environment after **LOCATE**. Listing the screen will automatically make it the current screen for editing (Figure Four-c).

Forth With a File System:
Kitt Peak VAX-Forth (11-NOV-82)

Here, **<BUILDS** is the word responsible for making headers. The definition of this word begins:

```
: <BUILDS  ?ALIGN LATEST  ,  ...
```

?ALIGN forces the dictionary pointer to the next longword boundary. Notice that Kitt Peak places the link field first, before the parameter field. This leads to the code in Figure Five-b.

Kitt Peak VAX-Forth uses the file system of the underlying operating system. Files which are being used are assigned positions in a file descriptor block table containing the file name, number of blocks, and status flags for each file. **LUN** is a variable holding the logical unit number (position in the table) of the currently active file. **BLK** contains the screen number of the current block within the current file. Both the logical unit number and the block number are saved (Figure Five-c).

BLK is -1 when the system reads from the keyboard. We save and restore only the current file. Kitt Peak's file loading word **LF** can be used to load a file. Its normal action is to make the file to be loaded the current file, load the file starting at block zero, close the file and restore the previous file. **LOCATE** requires that any file that has been used to define new words remain open. This necessitates that **LF** be modified not to close files and that both **LF** and **LOAD** be modified to set a file's flags so that the file cannot be closed. The redefinitions can be found in Figure Five-d. The only other change is that **THEN** must be replaced by **ENDIF** in **LOCATE**.

```
: WORD.        CR 5 SPACES  HERE COUNT TYPE  ;
: NOT-FOUND    WORD.  ."  is not in the dictionary " ;
: TOO-EARLY    WORD.  ."  is defined before LOCATE " ;
: KEYBD        WORD.  ."  was defined from the keyboard " ;

HERE CONSTANT WALL
: LOCATE   SAVE-ENVIRONMENT  FIND
              ?DUP 0=   IF  NOT-FOUND          ELSE
          DUP WALL  U<  IF  TOO-EARLY    DROP  ELSE
      CFA>SFA DUP KB?   IF  KEYBD        DROP  ELSE
                            SHOW-SCREEN   THEN THEN THEN
              RESTORE-ENVIRONMENT  ;
```

Figure One

```
: XCREATE   BLK @ ,  -FIND  ;
FIND XCREATE  ' CREATE !

: CFA>SFA   2+ NFA 2-  ;
: KB?  ( sfa -- true if keybd )  @ 0=  ;
: SHOW-SCREEN   ( sfa  --  )   @  LIST ;
: SAVE-ENVIRONMENT   ;
: RESTORE-ENVIRONMENT  ;
```

Figure Two — Installation of **LOCATE** in fig-FORTH

```
: VARIABLE   CREATE 0 ,  ;
: CONSTANT   HEAD,  <constant-cfa> CF,  ,  LINKLAST  ;
: CREATE     HEAD,  <variable-cfa> CF,  LINKLAST  ;
```

Figure Three-a — Decompilation

```
: HEAD,  BL WORD  DUP C@  .......

: XHEAD,   BLK @ , BL ;  ( patch into existing word )
    FIND XHEAD,  ' HEAD,  !
```

Figure Three-b — Partial definition of MMS's **HEAD,**

```
: CFA>SFA   8 - ;
: KB?  ( sfa -- true if keybd )  @ 0=  ;

: SHOW-SCREEN   ( sfa  --  )   @  LIST ;
  1 DARRAY SVDATA   VARIABLE SVSCR
: SAVE-ENVIRONMENT 2 0 DO    I BUFFDATA 2@ I SVDATA 2!
                        LOOP  SCR @ SVSCR !  ;
: RESTORE-ENVIRONMENT   SCR @  SVSCR @ SCR !
          0 BUFFDATA @ =  IF 0 1  ELSE 1 0 THEN
          BUFFDATA 2+ 1 SWAP C!
          DUP BUFFDATA 2+ 2 SWAP C!
          SVDATA @ DUP 0< NOT IF BLOCK THEN DROP
          2 0 DO  I SVDATA 2@  I BUFFDATA 2!  LOOP  ;
```

Figure Three-c

```
: SAVE-ENVIRONMENT   SCR @  SVSCR !  ;
: RESTORE-ENVIRONMENT  SVSCR @  DUP SCR !  BLOCK DROP ;
```

Figure Three-d

```
: VARIABLE   CREATE  2 ALLOT  ;
: CONSTANT   CREATE  ,  ;CODE  ...
: CODE  CREATE SMUDGE HERE  DUP 2- !  ENTERCODE  ;
```

Figure Four-a — Decompilation

```
: CREATE   BL WORD  DUP DUP  ....

: XCREATE   BLK @ , BL ;  ( patch into existing word )
  FIND XCREATE  ' CREATE !
```

Figure Four-b — Partial definition of MVP's CREATE

```
: CFA>SFA   2+  NFA  2-  ;
: KB?  ( sfa -- true if keybd )  @ 0=  ;
: SHOW-SCREEN   ( sfa  --  )   @  LIST ;
: SAVE-ENVIRONMENT   ;
: RESTORE-ENVIRONMENT   ;
```

Figure Four-c

```
: VARIABLE  CONSTANT ;CODE ...
: CONSTANT  <BUILDS ,  ;CODE ...
: CODE      ?EXEC  <BUILDS '  ;  IMMEDIATE
```

Figure Five-a — Decompilation

```
: X<BUILDS  ?ALIGN LUN @ B, BLK @ B, 0 B, 0 B, ;
    FIND X<BUILDS  ' <BUILDS !
```

Figure Five-b — Making headers in Kitt Peak's file system

```
: CFA>SFA   IA +  PFA>NFA  8 - ;
: KB?  ( sfa -- true if keybd )  1+ B@  255 =  ;
: SHOW-SCREEN  ( sfa  --  )  DUP B@ LUN ! FILE? 1+ B@ LIST ;
: SAVE-ENVIRONMENT     LUN @  ;
: RESTORE-ENVIRONMENT  LUN !  ;
```

Figure Five-c

```
: "LF    LUN @ >R  1 FTU.BIT OR  DOFOPEN  ?FILE  LUN !
         CR  0 0 .LINE  0 LOAD
         LUN @ FFLAGS  DUP @  DUP FTU.BIT AND
         IF   FTU.BIT COM  AND  ENDIF
         3 COM AND  1 OR  SWAP !    <R  LUN !   ;
: LF   ( LF filename )  GFN "LF  ;
: LOAD ( # --  )  LUN @  FFLAGS DUP @  1 OR  SWAP ! LOAD  ;
```

Figure Five-d

Forth Resources via Modem

Gary Smith
Little Rock, Arkansas

Daily more individuals are joining the Forth community, and they bring a need for information. Fortunately, most have modems and supporting telecommunications software. For at the same time, more Forth information than ever before is available on private and vendor-supported electronic bulletin boards, and also on some electronic information services like CompuServe[1] and BIX[2].

The following information is intended to bridge the gap that presently exists between the user in need and the resource waiting to serve. I am satisfied that I have not listed all possible resources and in the dynamic world being discussed here, there are certain to be changes. If readers help to supply additional information, or pose specific questions about electronic sources of Forth-related information, this feature can be updated as required. Just write to *Forth Dimensions* in a letter separate from any other FIG business; and be sure to include your electronic address, if you have one!

My very sincere thanks to Jerry Shifrin of the East Coast Forth Board for his help in compiling and verifying much of the information published here. His board is included in the list, and is truly one of the best of the listed resources.

All that follows was accurate (or reasonably so) at the time this article was written. The author, *Forth Dimensions* and the Forth Interest Group assume no responsibility for any omissions, deletions or errors.

BULLETIN BOARDS

Name/Provider	Phone No.	Hours	Baud/Protocol	Comments
The FIG Tree	415/538-3580	24 hours 7 days/wk.	300 7 bits, even parity	CommuniTree BBS Messages only. Lots of info. Type 2 CRs to begin.
Thousand Oaks Technical BBS	805/492-5472	24 hours 7 days/wk.	parity	Files supported.
East Coast Forth Board	703/442-8695	24 hours 7 days/wk.	300,1200,2400 8, 1, none	Registration req., but free. Very full service. Much from others ported here.
The Forth Source	215/630-9149	2pm–8am EST 24 hrs. on weekend	300	Jon Day (sysop) Forth conf., but no up/down load of files.
LMI BBS	213/306-3530	6pm–9am PST 24 hrs. on weekend	300,1200,2400 8, 1, none	Wes Meier BBS Support to users by Laboratory Microsystems. Others NOT restricted from use.
Correct Software			300,1200	Currently down. Non-users of Correct products send $15.00 for registration to: Rt. 1, Box 140 Black Hawk, SD 57718
The WELL	415/332-6106	24 hrs. 7 days/wk.	300,1200,2400	UNIX system conference **NOTE: Use lower case! Gateway to USENET via GO TELE then choose #13. Have VISA/MC ready and type newuser (lower) Mike Ham is Forth Guru here.
Computer Language	415/957-9370	24 hrs. 7 days/wk.	300,1200	FIDO BBS Extensive message base. Many files.

Information Systems

CompuServe (Call 800-848-8990 for information.) CompuServe features three conferences with extensive Forth coverage. Summaries are as follows:

"CLM Conference" Similar to The WELL and *Computer Language*'s own BBS, and operated by *Computer Language* magazine. DL-7 is the Forth library.

Go CLM at the ! prompt.

"Computer Solutions, Inc." Emphasis on CSI products MacForth and MultiForth, but also lots of goodies and discussion for the general Forth crowd. DL-6 is the Forth library.

Go PSG-4 or Go FORTH at the ! prompt.

"Dr. Dobb's Journal" This is the newest entry and is not yet as active as the previous two. With Ray Duncan and Michael Ham as sysops, that will change! DL-6 is the Forth library.

Go DDJFOR at the ! prompt.

The Special Interest Groups for the Model 100 and Commodore also have some excellent machine-specific Forth items. Commodore has Blazin' Forth (a public-domain Forth-83 implementation) and M-100 has a public-domain Forth that supports Model 100s using the Chipmunk drive.

BIX (Call 800-227-2983 for information.) BYTE Information Exchange is *BYTE* magazine's conference system. One of the hundred or so sections is the Forth conference with Phil Wasson as sysop. This is a good source for hot-breaking news. There is a separate conference area for files.

JOIN FORTH at the : prompt.

There is also considerable Forth discussion in appropriate conferences such as 4TH.GEN.LANGS but it must be sought out.

References

1. CompuServe
5000 Arlington Centre Blvd.
Columbus, OH 43220
(or inquire at your local dealer)

2. BYTE Information Exchange
70 Main Street
Peterborough, NH 03458

Forth Source Formatter

John Konopka
Mitaka Shi, Japan

It is considered good form in any computer language to keep one's source code neatly formatted. There are various recommendations about when to indent, when to start a new line, etc. The problem with this is that it takes too much effort. It's like asking a teenager to keep his room clean: nice in theory, but difficult to implement. The problem is accentuated for Forth programmers editing existing source who are constrained to add code to the fixed size of one screen. While we don't yet have robots which will clean our rooms for us, we do have a robot which can make source code legible: the computer.

The program presented here will take the most compact, confusing source code as input and will turn out a cleanly formatted listing with new lines for every colon definition and proper indenting for structured constructs. A fill algorithm is used to prevent words from running off the right edge of the page. Parameters are available to adjust the width and length of the listing to any printer or terminal. Line feeds are counted so that form feeds can be generated at appropriate intervals.

In operation, the program works similarly to a word processor. Instead of adding "dot commands" to the source to cause various actions, special Forth words are interpreted as commands to automatically cause formatting of the text. The code for the main loop is in screen #7. It parses one word of text, then checks if this is a special word. If it is, then some formatting occurs; if not, the word is simply typed. To avoid storing strings of text and doing string comparisons, **FIND** is used to get one word of source then convert this to an address. This address is compared against the stored addresses by the word **??** when checking for special words. Rather than assume anything about what **FIND** does to a string or where it leaves it, the value of **>IN** is saved before invoking **FIND** and then is restored in **TYPE-IT** before parsing the word again with **WORD**.

Screen #2 contains the parameters controlling the dimensions of the listing. The constants **LINE-SIZE** and **MAX-LINES** control the width and page length respectively. By changing these, you can match the listing to your printer or screen size. The constant **OFFSET** determines how far the margin is moved whenever an indent occurs. A value of three seems good, but you might want to use two or even one if you work with a very narrow page. Variable **COLUMN** is the counter containing the current character distance from the left edge of the page. Variable **TAB** is the left margin measured in spaces from the left edge of the page. Variable **LINES** counts the number of lines typed since the last form feed.

The arrays of stored addresses are held in screen #3. An array is created for each action desired. For example, **IF** and **DO** both require an indent and a new line, so they are placed in the array called **INS**. If you define new branching words you can add them to one of these arrays. The first integer in the array says how many addresses are in the array.

INDENT, **OUTDENT** and **NEW-LINE** (defined in screen #4) move the left margin in the fashion suggested by their names. Two other special formatting words are defined in screen #6. These are **TYPE-TO-** and **TYPE-TO-END-OF-LINE**. The first emits words until it encounters the character it is given on the stack. It is used to list out a series of words such as those between quotes or parentheses. The second emits text until **>IN** is an integer multiple of 64. Its purpose is to list comments set off by a backslash. This second word is an exception in that it types directly out of the text buffer and it does not check string length before typing. Both of these words list text without moving the margin, even though the typed string contains words such as **DO**, **LOOP**, **IF**, **THEN**, etc.

Two other words, **CR?** and **TYPE-IT** (screens #4 and #5 respectively), are used to send all text to the output device (with the one exception noted above). **CR?** performs a carriage return and line feed only when needed, avoiding blank lines in the output. It also counts the number of line feeds emitted and emits a form feed when the number of lines exceeds the value in constant **MAX-LINES**. **TYPE-IT** restores **>IN** from temporary storage in **IX** and then parses a string with **WORD**. It compares the length of each word to be typed with the remaining space on the line, then invokes **CR?** if there is insufficient room to type the word. It's possible to type off the right edge of the page if there is still insufficient room to type the word even after invoking **CR?**. This can happen with deeply nested **IF ELSE THEN** clauses and long names.

```
Screen #2
\ F+L  Variables and Constants
VARIABLE COLUMN        \ Current column pointer.
VARIABLE TAB           \ Position of left margin.
VARIABLE #LINES        \ Count of lines typed since last FF.
VARIABLE FND           \ Flag indicating currently parsed word was
                       \ found to be a special formatting word.
VARIABLE IX            \ Temporary storage of value of >IN.

79 CONSTANT LINE-SIZE  \ Column width of output device.
65 CONSTANT MAX-LINES  \ Lines per page of output device.
 3 CONSTANT OFFSET     \ Size of indentation in columns.
34 CONSTANT A"         \ ASCII value of double quote.
41 CONSTANT RPAREN     \ ASCII value of right parenthesis.

FIND \ CONSTANT BSLASH
FIND  ( CONSTANT PAREN


Screen #3
\ F+L  Arrays of pointers to formatting words.
CREATE QUOTES  2 , FIND "       , FIND ."      ,

CREATE STARTS  2 , FIND :       , FIND CODE    ,

CREATE ENDS    3 , FIND ;       , FIND NEXT,   ,  FIND END-CODE ,

CREATE OUTS    5 , FIND UNTIL   , FIND LOOP    ,  FIND +LOOP    ,
                   FIND THEN    , FIND REPEAT  ,

CREATE INS     3 , FIND IF      , FIND DO      ,  FIND BEGIN    ,

CREATE IN+OUTS 4 , FIND ELSE    , FIND WHILE   ,
                   FIND DOES>   , FIND ;CODE   ,
```

Text interpretation is halted when the value of **>IN** exceeds 1023, indicating that all the text from the screen has been used. You can stop the listing on other conditions by modifying the word **DONE?** in screen #5.

This program is not infallible. It is designed to format "normal" source code. If given source with special words in unusual contexts, it will produce unusual results. Unpaired parentheses or quotes are the most troublesome. Using it on screen #3, for example, will cause unanticipated results. However, used judiciously, it should be a very helpful tool for you. I find it useful to help me understand the flow of code, particularly when there are nested **IF ELSE THEN** constructs. Because every colon definition is placed on a new line, this program is good for finding those small definitions that sometimes get tucked in the corner of a screen. This code is also very useful for tracking down mistakes, as a missing or redundant word such as **THEN** or **LOOP** causes things to skew unexpectedly. In valid code, every addition to **TAB** should be matched by a subtraction from **TAB** so that ; will fall in the same column as :. If an excess of **OUTDENT** calls tries to move the margin off the left edge of the page, then **TAB** is clamped at zero and **BELL** is invoked.

```
Screen #4
 \ F+L  CR? INDENT OUTDENT NEW-LINE
 \ ---  Perform a carriage return (maybe) and form feed (maybe).
 \      Don't do CR on new lines. This avoids blank lines.
 : CR? COLUMN @ TAB @ >                  \ Legitimate CR ?
      IF CR #LINES @ MAX-LINES =          \ Do CR, need a FF ?
        IF  SFF 0 #LINES !                \ Do FF
        ELSE 1 #LINES +!                  \ Count lines
        THEN 0 COLUMN ! THEN ;            \ Reset column pointer.
 \ n ---   Indent by constant OFFSET times n.
 : INDENT  CR?  OFFSET * TAB +! ;

 \ n ---   Move margin left OFFSET times n. Don't move off page.
 : OUTDENT CR? TAB @ SWAP OFFSET * - DUP 0<
           IF BELL THEN 0 MAX TAB ! ;
 \ ---     Reset left margin and invoke carriage return test.
 : NEW-LINE 0 TAB ! CR? ;


Screen #5
 \ F+L   DONE?   PARSE   TYPE-IT   SFF
 \  ---  fl   Check for end of screen.
 : DONE? >IN @ 1023 >= ;

 \ ---   Place next word at HERE ready for typing.
 : PARSE IX @ >IN ! BL WORD DROP ;

 \ ---   Type one word, first do CR if no room on this line.
 : <TYPE> COLUMN @ TAB @ < IF TAB @ COLUMN @ - SPACES TAB @
        COLUMN ! THEN HERE COUNT 1+ DUP COLUMN @ + LINE-SIZE >
        IF CR? TAB @ SPACES TAB @  COLUMN ! THEN DUP
        COLUMN +! TYPE -1 FND ! ;
 \ ---
 : TYPE-IT PARSE <TYPE> ;
 \ ---     Emit a form feed.
 : SFF 12 EMIT ;


Screen #6
 \ F+L  ?? F+L-INIT TYPE-TO- TYPE-TO-END-OF-LINE
 \ x y --- x fl  Flag true if x found in array at address y.
 : ??  0 SWAP DUP SWAP @ 0 DO 2 + DUP @ 4 PICK =
       IF SWAP 1+ SWAP LEAVE THEN LOOP DROP ;

 \ n --- m p  List block n. Save BLK and >IN, restore later.
 : F+L-INIT BLK @ >IN @ ROT BLK ! 0  >IN ! 1 #LINES !
       0 TAB ! 0 COLUMN !  ." Screen #" BLK @ . CR  ;

 \ n ---       Type text till next occurence of n.
 : TYPE-TO-  CR? TYPE-IT WORD C@ 1+  HERE C! BL HERE DUP C@
       1+ + C! <TYPE> CR? ;
 \---   Type comment line, round up >IN.
 : TYPE-TO-END-OF-LINE CR? TYPE-IT BLK @ BLOCK >IN @ + >IN @ 63
       COM AND 64 + DUP >R >IN @ - -TRAILING TYPE CR? R> >IN ! ;
```

```
Screen #7
 \ F+L  END
 \ n ---      Formatted listing of screen #n.
 : F+L    F+L-INIT
 BEGIN 0   FND ! >IN @ IX ! FIND
   QUOTES    ??  IF         A"  TYPE-TO-                   THEN
   DUP PAREN =  IF    RPAREN  TYPE-TO-                   THEN
   DUP BSLASH = IF            TYPE-TO-END-OF-LINE        THEN
   STARTS    ??  IF NEW-LINE  TYPE-IT >IN @ IX !
                              TYPE-IT 2 INDENT           THEN
   INS       ??  IF 1 INDENT  TYPE-IT 1 INDENT           THEN
   OUTS      ??  IF 1 OUTDENT TYPE-IT 1 OUTDENT          THEN
   ENDS      ??  IF 2 OUTDENT TYPE-IT CR?                THEN
   IN+OUTS   ??  IF 1 OUTDENT TYPE-IT 1 INDENT           THEN
   DROP DONE?    IF NEW-LINE  TYPE-IT CR SFF 1 ELSE 0 THEN
     FND @  NOT IF            TYPE-IT                    THEN
 UNTIL  >IN ! BLK ! ;
```

```
: EXAMPLE X1 X2 = IF Y2 Y3 AND ELSE X3 IF \ DO Z10 IF X3 TRUE
 Z10 ELSE ERROR THEN THEN ; : TEST 10 X1 ! ( Std Val) EXAMPLE ;
: FINAL ( Run Experiment) INIT-ADC TURN-ON-TESTER EXAMPLE ;
```

Source code before formatting.

```
Screen #8
: EXAMPLE
      X1 X2 =
         IF
            Y2 Y3 AND
         ELSE
            X3
               IF
                  \ DO Z10 IF X3 TRUE
                  Z10
               ELSE
                  ERROR
               THEN
         THEN
;
: TEST
      10 X1 !
      ( Std Val)
      EXAMPLE
;
: FINAL
      ( Run Experiment)
      INIT-ADC TURN-ON-TESTER EXAMPLE
;
```

Source code after formatting.

Dual-CFA Definitions

Mike Elola
San Jose, California

Decomposing functions is a critical part of Forth programming. Dual-CFA definitions provide the benefits of decomposed functions in situations where decomposition would not be possible normally. Besides requiring two CFAs, this kind of functional decomposition is implemented in reverse of the normal dictionary order. (Normally, Forth requires a bottom-up ordering of definitions — child definitions are compiled ahead of the parent definitions that reference them.)

By first exploring more conventional Forth programming techniques, various issues will emerge that showcase the advantages of dual-CFA decomposition.

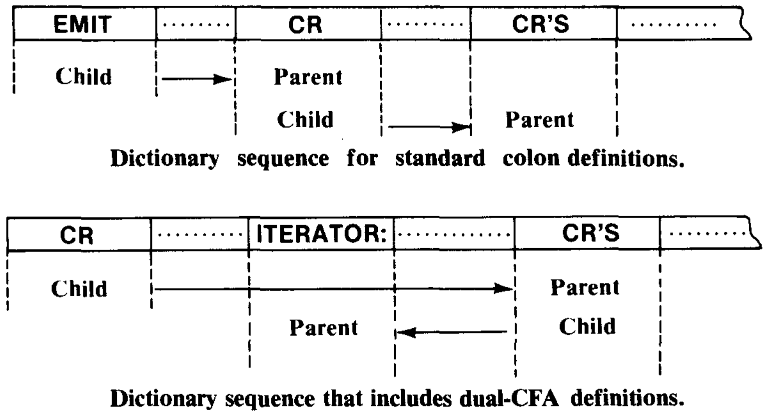


Figure One

Illustrative Example

Many dual-CFA definitions could be replaced with similar definitions that use vectored execution. Unfortunately, the use of execution vectors can negate certain advantages of normal Forth decomposition.

Suppose that you have revectored **TYPE** to route its output to the printer. A pause is desired to allow users to insert another sheet of paper, so you define the following word:

```
: FEED-PRINTER   ( -- )
CR ." Insert new sheet. Press any key to continue."
KEY DROP  ;
```

When run, the message is typed at the printer rather than displayed at the user terminal. **TYPE** is called by **(.")** which is compiled by ." within the definition of **FEED-PRINTER**.

The source of this problem may be the decomposition itself. The revectored version of **TYPE** shows distinct I/O redirection behavior. In normal development, the enlargement of **TYPE**'s function would be implemented in a new and separate definition incorporating **TYPE**. Then, higher-level words such as **FEED-PRINTER** could engage the correct version of **TYPE** upon compilation. There would be no possible chance for erroneous, surprise behavior.

Including the function of output redirection within **TYPE** integrates more functionality into the word than was originally intended. This prevents a more complete factoring of these related but distinct functions. As such, **TYPE** is a poorly decomposed function. Because of the use of a vector, less consideration than usual is given to the decomposition involved. (This is something to stay on guard for when execution vectors are used in an application.)

Compensatory code can be formatted to solve the problems associated with this example. For instance, **FEED-PRINTER** can buffer the contents of the vector variable, reset it to a value that causes output to the screen, write the message, then reinitialize the original vector value.

However, such remedies add complexity to the task of programming and increase the chance of programming error. When the behavior of a word must be variable, extra code is required each time it is used to ensure that the correct behavior will be selected when it is run. Ultimately, a heavy price is paid when compiling words with variable (loosely-coupled) actions.

This example is not presented to suggest that vectored execution is never advantageous. The run-time flexibility of vectored execution is justification enough for its use. But when the desired behavior can be selected at compile time, a better programming solution is possible.

Preferred Forth Style

Forth is at its best when variable-behavior definitions are avoided. Functions have to be well decomposed when fixed-behavior words are used exclusively. Together, such words impart hardiness to an application by eliminating dependence on the current environment.

As an extensible programming language, Forth can exhibit a wide range of functionality: with each new word added, its functionality is extended. In this way, Forth can provide a wide range of functions to better deal with a wide range of programming problems. If the extensions are fixed-behavior words, Forth offers superior ease of programming as well.

A Forth programming philosophy aimed at memory compactness, brevity of expression and ease of use is realized when words in higher memory consistently integrate more functionality than those located in lower memory. Lower-level words should exhibit decreasing functional scope. All words should perform single, fixed behaviors.

To achieve this, each new higher-level definition should consolidate more functionality. This way, each word in a common execution path represents an increment of progress toward the overall application function. Progress toward the ultimate goal that is not incremental may create problems and usually indicates incomplete decomposition.

Programs developed in this way exhibit the following characteristics:

1. Compactness of compiled code
2. Fewer conditional phrases
3. Single behaviors per word

The following three sections explore the effectiveness of other, more conventional techniques that in some ways comply or do not comply with this programming philosophy.

Deferred Definitions

Normally, the behavior of a deferred word is not intended to be variable. Unanticipated behavior is only possible when deferred words are incorrectly initialized. Once initialized, deferred words should provide the same benefits as fixed-behavior words.

To ascertain that one word represents uniformly increasing functionality relative to others, it is necessary to examine its relation to other words along the same execution path. (It is not relevant simply to compare a word's physical location with that of its parents.) Consequently, deferred words and dual-CFA words can both be used effectively to create a sequence of words where each subsequent one consolidates slightly more functionality.

Vectored Definitions

Frequently, vectored definitions are employed to create words with variable behaviors. As such, their use should be avoided if at all possible. The first example given (**FEED-PRINTER**) helped demonstrate the problems that arise when vectored definitions disguise poor decomposition. Still other problems are directly associated with the use of vectored execution (as well as with other variable-behavior definitions).

While vectored definitions can execute rapidly, they also create additional work. An effort must be made to maintain the correct "current" value of a vector, particularly when it is dynamically changing. This usually takes the form of "housekeeping" initializations and finalizations with each use of the vectored word. The extra code can obscure the underlying algorithm.

Another problem with vectored definitions is increased vulnerability to error — particularly when one revectoring word calls another. Unless a stack is used to help maintain the correct value of the execution vector, the higher-level word has no guarantee that its initializations of vectors remain in effect after other words are called.

Still, memory compactness is sometimes served best through vectored execution. Redirection of input or output is a good example of this. Revectoring **EMIT** is more memory efficient than generating custom versions of **EMIT**, **CR**, **SPACE** and **TYPE** for every possible output device.

While vectoring enables run-time flexibility and seems to be the shortest path to a coded solution, the support required to deal with the added complexity must be considered. A variable-behavior **EMIT** may create the need for fixed-behavior versions of output words just to make programming easier. For instance, if there is a need for **FEED-PRINTER**, then it alone is justification for **SCREEN-TYPE**, a fixed-behavior word. Ultimately, a mix of fixed- and variable-behavior words may prove most effective in situations like this.

Other Variable-Behavior Definitions

The behavior-binding function of the Forth compiler is rendered inadequate whenever variable word behaviors are allowed. This is true however the variable behaviors are ultimately selected. Vectored definitions are one form of variable-behavior words. Other kinds of words are also used to implement variable behaviors.

A way to select one of several actions is through the use of a conditional phrase. Consider the following definition for **TYPE**:

```
: TYPE  ( add count -- )
  OUT-DEV  @ IF  ( printer code... )
  ELSE  ( screen code... )  THEN  ;
```

The behavior of this word is under the control of the variable **OUT-DEV**. Now, before and after each use of **TYPE** the programmer must decide how **OUT-DEV** should be initialized and finalized. Note that the same burdens of environmental maintenance befall the use of this definition as would a vectored definition. Only now, **OUT-DEV** is the environmental variable requiring extra care, instead of a vector-containing variable.

The following is an attempt to improve on the previous version:

```
: TYPE  ( add count dev -- )
  IF  ( printer code... )
  ELSE  ( screen code... )  THEN  ;
```

Now the problems with maintaining the current environment disappear because the behavior-selection mechanism is one of **TYPE**'s input parameters. The drawbacks that remain are the extra execution overhead and memory overhead for the conditional phrase, as well as the extra code necessary to generate the additional stack item consumed with every call to **TYPE**. All of these drawbacks could be avoided if a fixed-behavior, screen-oriented **TYPE** were available (assuming that it could be selected at compile time).

(When run-time flexibility is needed, there is often no choice but to use a variable-behavior word. Such is the case when user-selectable actions are desired. The Forth interpreter manages to accomplish this admirably, since it is very compact and employs very few conditionals.)

Dual-CFA Definitions

Deferred definitions, vectored definitions and other variable-behavior definitions have been considered so far. However, dual-CFA definitions have yet to be separately presented. For the purpose of discussion, suppose that a definition of **ITERATOR:** already exists, and that it is a defining word that can be used to produce dual-CFA definitions. The first CFA compiled points at an iteration function that can be the parent to many similar words, such as **CR'S** and **SPACES**:

```
ITERATOR:  CR'S CR  ;
ITERATOR:  SPACES SPACE  ;
```

An equivalent colon definition for **CR'S** is:

```
: CR'S   0 DO CR LOOP  ;
```

These definitions will help illustrate the merits of dual-CFA definitions in terms of the programming philosophy already presented.

The dual-CFA definitions are obviously much shorter. This is possible because the common code for **CR'S** and **SPACES** has been factored into the parent defining word **ITERATOR:**.

The compactness of the compiled code for **CR'S** and **SPACES** is also an indication that these definitions are rooted in a consistent philosophy of decomposition.

Since all behaviors have been fixed at compile time, no intermediary variables are required. This makes the use of **CR'S** and **SPACES** as straightforward as can be; there are none of the added complexities described for vectored and variable-behavior definitions.

To illustrate that the dual-CFA implementation represents more incremental decomposition, the child-to-parent execution sequence of the dual-CFA definition is:

```
CR --> CR'S --> ITERATOR:
```

The child-to-parent execution sequence of the corresponding colon definition is:

```
CR --> CR'S
```

The dual-CFA definition exhibits more incremental decomposition (three call levels, opposed to two).

These execution sequences may not be understood without further explanation. Although **ITERATOR:** is the parent of **CR'S**, its run-time code is not callable except through the child definition. The invocation of the child word **CR'S** remains the starting point for the execution sequence. Its first CFA calls **ITERATOR:** which in turn calls the second CFA of **CR'S**, which executes the code specifically compiled for **CR'S**. This should become clearer when shown in dictionary order as in Figure One.

Note also that the resulting dictionary order is consistent: words in higher memory integrate more functionality than those in lower memory. For example, **CR'S** belongs at a higher position than **ITERATOR:** because it integrates more functionality. **ITERATOR:** is also positioned correctly — ahead of both **CR'S** and **SPACES** — so that it can be a reuseable behavior. This ordering of definitions corresponds to a decomposition of functions that can best leverage one's programming effort.

One disadvantage associated with the dual-CFA implementation is the challenge posed when trying to formulate the parent defining word. The second CFA of the child will always be the top item on the stack when the parent code executes. For this and other reasons, the parent definitions are more difficult to formulate than normal definitions.

Implementation Details

Providing an easy means for compiling dual-CFA definitions is another concern. The Forth resource best suited to the task is the **CREATE DOES>** combination.

Listing One shows the necessary support words needed for both Forth-79 and Forth-83. These words can be used to create a dual-CFA defining word such as the one suggested in the previous examples:

```
ITERATOR:
  CREATE  DOCOL  ,  COMPILE-DEF
  DOES>
    SWAP 0  ( cfaz #times 0 -- )
    DO DUP EXECUTE LOOP  DROP ;
```

Other support words must be added to allow recursion within the parent portion of dual-CFA definitions. Still other support words, or altogether different implementations, are needed to support multi-level, dual-CFA decomposition.

Other dual-CFA defining words I have found useful include:

TREE-TRAVERSER: (with **TREE-BALANCE**, **TREE-SCAN** and **TREE-PRINT**), **EXEC-COUNT:**, **CALL-TRACE:**, and **RELOCATABLE:**.

```
: COMPILE-DEF  ( Forth-83 )
  !CSP  SMUDGE [COMPILE]  ]  ;

: COMPILE-DEF  ( Forth-79 )
  SP@  CSP  !
  SMUDGE  [COMPILE]  ]  ;

: COMPILE-DEF  @ CONSTANT DOCOL
( Forth-83 )

: COMPILE-DEF  CFA @ CONSTANT DOCOL
( Forth-79 )
```

Listing One — Dual-CFA Support Words

Conclusions

The Forth language suggests a philosophy of decomposition that rewards well those who find ways to conform with it. Compliance involves compiling fixed-behavior words. Each new word is defined to consolidate slightly more functionality than the previous one in the same execution path.

Dual-CFA definitions can uphold this philosophy when other kinds of definitions do not. Particularly in those situations where a deviation in dictionary sequence is required, this technique reaps all the benefits normally reserved for Forth words implemented in the standard child-before-parent sequence.

The Forth language generates enthusiasm among programmers by offering the possibility of very streamlined code. Considering the prospects for dual-CFA decomposition, this enthusiasm can be expected to increase.

(Letters continued from page 8)

Mr. Apra's suggestion for experiments with conditional structures as a source of programming exercises for students.

Forth offers a unique choice: one can choose programming languages that incorporate a large number of "features" (which can't be changed and which may be hard to use or to remember) or one can choose a programming language which is (conceptually) simple and allows the user to incorporate a wide variety of features on his own. Understanding is a powerful "handle" which I would not want to trade for "features." I echo Mr. Apra's "Keep it simple," but add the words "... in concept."

Sincerely yours,

John J. Wavrik
U.C. San Diego, California

Index to Advertisers

FIG Chapters

U.S.

• ALABAMA

Huntsville FIG Chapter
Call Tom Konantz
205/881-6483

• ALASKA

Kodiak Area Chapter
Call Horace Simmons
907/486-5049

• ARIZONA

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

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

• ARKANSAS

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

• CALIFORNIA

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

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

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

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

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

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

Bay Area Chapter
Silicon Valley Chapter
Monthly, 4th Sat.
FORML 10 a.m., Fig 1 p.m.
H-P Auditorium
Wolfe Rd. & Pruneridge,
Cupertino
Call John Hall 415/532-1115
or call the FIG Hotline:
408/277-0668

Stockton Chapter
Call Doug Dillon
209/931-2448

• COLORADO

Denver Chapter
Monthly, 1st Mon., 7 p.m.
Cliff King
303/693-3413

• CONNECTICUT

Central Connecticut Chapter
Call Charles Krajewski
203/344-9996

• FLORIDA

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

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

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

• GEORGIA

Atlanta Chapter
3rd Tuesday each month, 6:30 p.m.
Computone Cottilion Road
Call Ron Skelton
404/393-8764

• ILLINOIS

Cache Forth Chapter
Call Clyde W. Phillips, Jr.
Oak Park
312/386-3147

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

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

Rockwell Chicago Chapter
Call Gerard Kusiolek
312/885-8092

• INDIANA

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

Fort Wayne Chapter
Monthly, 2nd Tues., 7 p.m.
IPFW Campus
Rm. 138, Neff Hall
Call Blair MacDermid
219/749-2042

• IOWA

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

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

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

• KANSAS

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

• LOUISIANA

New Orleans Chapter
Call Darryl C. Olivier
504/899-8922

• MASSACHUSETTS

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

• MICHIGAN

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

• MINNESOTA

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

• MISSOURI

Kansas City Chapter
Monthly, 4th Tues., 7 p.m.
Midwest Research Institute
MAG Conference Center
Call Linus Orth
913/236-9189

St. Louis Chapter
Monthly, 1st Tues., 7 p.m.
Thornhill Branch Library
Contact Robert Washam
91 Weis Dr.
Ellisville, MO 63011

• NEVADA

Southern Nevada Chapter
Call Gerald Hasty
702/452-3368

• NEW HAMPSHIRE

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

• NEW MEXICO

Albuquerque Chapter
Monthly, 1st Thurs., 7:30 p.m.
Physics & Astronomy Bldg.
Univ. of New Mexico
Jon Bryan
Call 505/298-3292

• NEW YORK

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

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

Rockland County Chapter
Call Elizabeth Gormley
Pearl River
914/735-8967

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

• OHIO

Akron Chapter
Call Thomas Franks
216/336-3167

Athens Chapter
Call Isreal Urieli
614/594-3731

Cleveland Chapter
Call Gary Bergstrom
216/247-2492

Cincinatti Chapter
Call Douglas Bennett
513/831-0142

Dayton Chapter
Twice monthly, 2nd Tues., &
4th Wed., 6:30 p.m.
CFC 11 W. Monument Ave.
Suite 612

Dayton, OH
Call Gary M. Granger
513/849-1483

• OKLAHOMA

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

• OREGON

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

• PENNSYLVANIA

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

• TENNESSEE

East Tennessee Chapter
Monthly, 2nd Tue., 7:30 p.m.
Sci. Appl. Int'l. Corp., 8th Fl.
800 Oak Ridge Turnpike, Oak Ridge
Call Richard Secrist
615/483-7242

• TEXAS

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

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

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

Periman Basin Chapter
Call Carl Bryson
Odessa
915/337-8994

• UTAH

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

• VERMONT

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

• VIRGINIA

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

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

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

• WISCONSIN

Lake Superior FIG Chapter
Monthly, 2nd Fri., 7:30 p.m.
University of Wisconsin
Superior
Call Allen Anway
715/394-8360

Milwaukee Area Chapter
Call Donald H. Kimes
414/377-0708

MAD Apple Chapter
Contact Bill Horzon
129 S. Yellowstone
Madison, WI 53705

FOREIGN

• AUSTRALIA

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

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

• BELGIUM

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

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

• CANADA

Alberta Chapter
Call Tony Van Muyden
403/962-2203

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

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

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

• COLOMBIA

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

• ENGLAND

Forth Interest Group — U.K.
Monthly, 1st Thurs.,
7p.m., Rm. 408
Polytechnic of South Bank
Borough Rd., London
D.J. Neale
58 Woodland Way
Morden, Surry SM4 4DS

• FRANCE

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

• GERMANY

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

• HOLLAND

Holland Chapter
Contact: Adriaan van Roosmalen
Heusden Houtsestraat 134
4817 We Breda
31 76 713104

FIG des Alpes Chapter
Contact: Georges Seibel
19 Rue des Hirondelles
74000Annely
50 57 0280

• IRELAND

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

• ITALY

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

• JAPAN

Japan Chapter
Contact Toshi Inoue
Dept. of Mineral Dev. Eng.
University of Tokyo
7-3-1 Hongo, Bunkyo 113
812-2111 ext. 7073

• NORWAY

Bergen Chapter
Kjell Birger Faeraas
Hallskaret 28
Ulset
+47-5-187784

• REPUBLIC OF CHINA

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

• SWEDEN

Swedish Chapter
Hans Lindstrom
Gothenburg
+46-31-166794

• SWITZERLAND

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

SPECIAL GROUPS

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

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

FIGGRAPH
Call Howard Pearlmutter
408/425-8700

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

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

Address Correction Requested