$10 Volume XVI, Number 3 September 1994 October

FORTH DIMENSIONS

The Essence of Forth

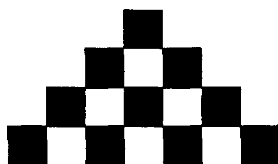
"Switch" in Forth

File-Format Sleuth

Engineering Notation

Mouse-and-Button Words

Interactive Target Compilation

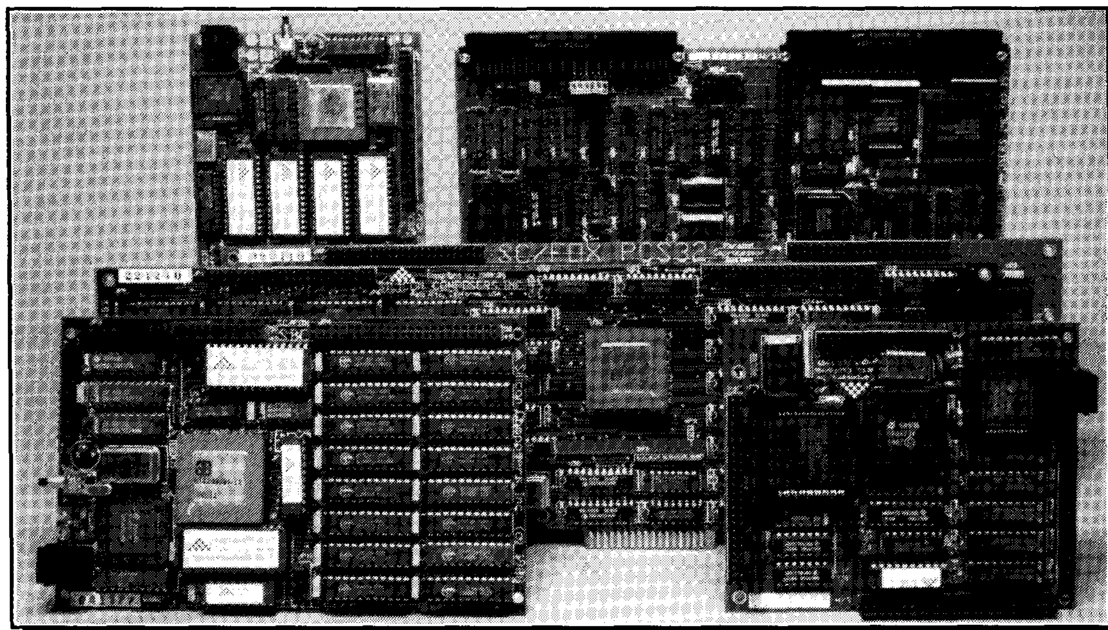


SILICON COMPOSERS INC

FAST Forth Native-Language Embedded Computers

DUP

>R



C@

R>

Harris RTX 2000™ 16-bit Forth Chip
- 8 or 10 MHz operation and **15 MIPS** speed.
- 1-cycle 16 x 16 = 32-bit multiply.
- 1-cycle 14-prioritized interrupts.
- two 256-word stack memories.
- 8-channel I/O bus & 3 timer/counters.

SC/FOX PCS (Parallel Coprocessor System)
- RTX 2000 industrial PGA CPU; 8 & 10 MHz.
- System speed options: 8 or 10 MHz.
- 32 KB to 1 MB 0-wait-state static RAM.
- Full-length PC/XT/AT plug-in (6-layer) board.

SC/FOX VME SBC (Single Board Computer)
- RTX 2000 industrial PGA CPU; 8, 10, 12 MHz.
- Bus Master, System Controller, or Bus Slave.
- Up to 640 KB 0-wait-state static RAM.
- 233mm x 160mm 6U size (6-layer) board.

SC/FOX CUB (Single Board Computer)
- RTX 2000 PLCC or 2001A PLCC chip.
- System speed options: 8, 10, or 12 MHz.
- 32 KB to 256 KB 0-wait-state SRAM.
- 100mm x 100mm size (4-layer) board.

SC32™ 32-bit Forth Microprocessor
- 8 or 10 MHz operation and **15 MIPS** speed.
- 1-clock cycle instruction execution.
- Contiguous 16 GB data and 2 GB code space.
- Stack depths limited only by available memory.
- Bus request/bus grant lines with on-chip tristate.

SC/FOX SBC32 (Single Board Computer32)
- 32-bit SC32 industrial grade Forth PGA CPU.
- System speed options: 8 or 10 MHz.
- 32 KB to 512 KB 0-wait-state static RAM.
- 100mm x 160mm Eurocard size (4-layer) board.

SC/FOX PCS32 (Parallel Coprocessor Sys)
- 32-bit SC32 industrial grade Forth PGA CPU.
- System speed options: 8 or 10 MHz.
- 64 KB to 1 MB 0-wait-state static RAM.
- Full-length PC/XT/AT plug-in (6-layer) board.

SC/FOX SBC (Single Board Computer)
- RTX 2000 industrial grade PGA CPU.
- System speed options: 8, 10, or 12 MHz.
- 32 KB to 512 KB 0-wait-state static RAM.
- 100mm x 160mm Eurocard size (4-layer) board.

For additional product information and OEM pricing, please contact us at:

SILICON COMPOSERS INC 655 W. Evelyn Ave. #7, Mountain View, CA 94041 (415) 961-8778

Contents

Features

7 ***File Format Sleuth*** *Bradley R. Olson*

Ten years ago, file formats were no mystery. But now, data is hidden under objects, file formats are hidden under applications, and hardware is hidden under an API. When you really need to know how an application puts together the files it generates, this "format ferret" will show you the internals in a way you can understand.

15 ***Engineering Notation with Integer Math*** *Richard W. Fergus*

Standard Forths are very satisfactory for accumulating and massaging data but may not provide an appropriate format for data display in certain applications. The format should account for data range *and* precision. The author's definitions in standard Forth provide engineering notation with selectable significant digits, decimal point positioning, and exponent offset.

18 ***Interactive Remote Target Compilation*** *Alan M. Robertson*

In these days of fat Forths and heavyweight hardware, some companies still thrive in a minimalist landscape. The author shares his techniques for working in 2K of code space, with an eight-deep stack and 35 bytes of RAM, showing how a little RISC can pay off.

21 ***Switch in Forth*** *Walter J. Rottenkolber*

Most Forthwrights are familiar with CASE to replace multiple branching statements. Its C equivalent, Switch, is less commonly seen. So, for frustrated C mavens, the incurably curious, and the enlightened who are converting C to Forth, herewith: Switch in Forth.

23 ***The Essence of Forth...*** *Randy Leberknight & Dennis Ruffer*

Programmers at the grande dame of Forth vendors are investigating Forth systems of the future. Here they remind us of the long-standing observation that Forth has an almost magical effect on productivity. But why? And how do Forth systems designers meet the requirements of the future without sacrificing the unique leverage that the language offers?

26 ***Simple Mouse and Button Words*** *Richard C. Wagner*

DOS-based Forth systems too often lack—unnecessarily—the ubiquitous, point-and-click graphical user interface. But you *can* have such tools, even without a Windows-based Forth. This type of environment can be supplied with only ten blocks of code. The system presented here provides words to communicate with the mouse drivers, and to display the buttons, detect a button "press," and execute the code associated with a button.

Departments

4 ***Editorial*** .... A gauntlet tossed—choose your weapons.

5 ***Letters*** .... The Scientific Forth Library Project; Yes, Virginia; Producing correct code; Random erratum; Pictures worth a thousand comments.

22 ***Advertisers Index***

38 ***Fast Forthward*** .... Exposing Forth's modules; correction to ANS Forth Quick Reference Card.

Editorial

A Gauntlet Tossed...

I'd like to call your attention to the Scientific Forth Library project announced in the "Letters" section. Not only is this an excellent opportunity for members of the Forth community to demonstrate leadership, it also points out what organized, motivated individuals can accomplish.

The Forth Interest Group, as a small non-profit organization, does not have the budgetary means to conduct market research or to hire project coordinators. Therefore, it often serves best by encouraging, facilitating, and coordinating the efforts of self-motivated (and usually self-appointed) leaders.

Without Bell Labs, would Unix even exist? Would it have made the transition from an interesting, experimental idea to widespread adoption and commercial success?

To those who bemoan Forth's limited penetration, I reply that Forth has enjoyed spectacular success, given its origins and relative lack of academic or corporate sponsorship. Besides, playing the victim role doesn't change anything—at least, not for the better.

What *can* boost Forth's viability and public esteem is leadership. FIG either has no R&D and no public relations, or it has a volunteer staff of a thousand inspired, talented, and motivated workers doing those things. The difference lies in how its members view themselves and their relationship to the organization.

The new ANS Forth standard is an example of what a dedicated and persistent group of Forth users can achieve. The "Top Ten List" explains good, succinct reasons for adopting Forth, and is available only because Mike Elola tackled the job and arranged with FIG for its use at trade shows, in *Forth Dimensions*, and in vendor-distributed literature. The high-powered FORML Conference is a resounding annual success because of individual leadership. The Scientific Forth Library will come to fruition because (a) someone had the idea and built others' enthusiasm for it, (b) vendors mindful of the opportunity will support an interface to existing Fortran code, (c) a number of programmers will cooperate to develop and refine an extensive library of stable, useful routines whose (d) widespread availability will bring more users to FIG and to commercial Forth systems that support the concept.

Those are a few examples, of varying scale, demonstrating the merits of coordinated volunteerism. The best-organized FIG chapters are another case in point. Of course, the very availability of Forth today is due to Charles Moore's personal inventiveness; and FIG's existence sprang from the original, do-it-because-it's-a-good-idea, fig-Forth implementation teams.

Choose Your Weapons

The most successful volunteer efforts appear to come from someone who is inspired enough to make a project "their own" while staying flexible enough to consider the needs of the community. So the idea you originate, the concept that really "speaks" to you, may be the one to which you can best contribute. But here are a few entries from my personal list of projects that need one or more dedicated champions:

(Continues on page 36.)

Forth Dimensions
Volume XVI, Number 3
September 1994 October

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $40 per year ($52 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 2154, Oakland, California 94621. Administrative offices: 510-89-FORTH. Fax: 510-535-1295. Advertising sales: 805-946-2272.

Copyright © 1994 by Forth Interest Group, Inc. The material contained in this periodical (but not the code) is copyrighted by the individual authors of the articles and by Forth Interest Group, Inc., respectively. Any reproduction or use of this periodical as it is compiled or the articles, except reproductions for non-commercial purposes, without the written permission of Forth Interest Group, Inc. is a violation of the Copyright Laws. Any code bearing a copyright notice, however, can be used only with permission of the copyright holder.

The Forth Interest Group
The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. Many research hardware and software designs that will advance the general state of the art. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $40/46/52 per year by the Forth Interest Group, 4800 Allendale Ave., Oakland, CA 94619. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621-0054."

Letters

Letters to the Editor—and to your fellow readers—are always welcome. Respond to articles, describe your latest projects, ask for input, advise the Forth community, or simply share a recent insight. Code is also welcome, but is optional. Letters may be edited for clarity and length. We want to hear from you!

Yes, Virginia (well, maybe...)

Dear Marlin,

About two months ago, I notified the FIG office of an opening in Houston for a Forth programmer. I learned of the opening from a programmer who works for the company having the opening. When I contacted the FIG office a month later regarding another matter, I asked how many programmers FIG had referred to the Houston company. To my surprise, I was told that the opening had not been posted, and that it was FIG policy not to post a listing unless FIG had received written notification of the opening from the company having the opening. To do otherwise, I was told, would jeopardize the credibility of FIG.

Wake up, FIG! In the current job market, most advertised openings are filled within a day or two. Although the FIG office contends that there are few, if any, west-coast Forth programmers needing work, I have personally heard from a number of Forth programmers throughout the nation who have been unable to find work and are rapidly becoming desperate. A person needing work is generally willing to follow any promising lead, and he will seldom complain if the lead doesn't pan out. It is unconscionable for FIG to refuse to pass on a lead, unless the lead is obviously bogus.

Working at what you are good at and want to do is where the money and return are.

So FIG is trying to figure out how to attract new members and how to hold onto current members? Obviously, a change in policy is indicated.

- Spend perhaps two thousand dollars for a '386 clone, a good modem, and a phone line.
- Begin operating a computer bulletin board, calling it "The FIG Job Board." If no one in the FIG office can serve as sysop, surely a FIG member will volunteer: just ask.
- *Prominently* publicize the board in each and every issue of *Forth Dimensions*—devote at least half a page.

The Scientific Forth Library Project

The Forth language is at an important crossroad with regard to its use as a general scientific programming language. The new FORTRAN-90 is just becoming available, and long-time FORTRAN programmers are finding it different enough that many are wondering if they might as well learn a new language instead of sticking with FORTRAN. If the Forth community plays its hand right, that alternate language could be Forth. To do so, Forth needs to overcome the standard complaints of the FORTRAN community:

1. It's not standardized, so how can I port my software?
2. I have lots of pre-existing FORTRAN code that works perfectly well, and I am not in any hurry to re-write it. Can Forth interface with my FORTRAN code?
3. There are no scientific libraries in Forth.

The recently adopted ANS Forth handily addresses #1, and in fact it is the adoption of the standard that makes issues #2 and #3 worth addressing.

With regard to #2, I think the adoption of the standard will help, since the interface to other software is the kind of feature that will distinguish one vendor's ANS Forth from another's. While the standard does not address such interfaces, I don't think there will be too much divergence on how this is done. The Unix world has no such standard, and I have only encountered two different C-FORTRAN conventions in over 15 years of using Unix.

So #1 is now solved, and the vendors will (I hope!) address #2. The third point can be addressed by the Forth community itself. Several potential scientific users of Forth discussed these issues at the recent Rochester Forth Conference. It was decided that we should undertake the project of writing a scientific library in ANS Forth.

The plan is to write a set of Forth words to implement such libraries as the ACM's BLAS, LINPACK, etc. The libraries will be publicly available in source form (in some sort of "public" release: public domain, copyleft, copyrighted but freely distributable, etc.).

To get started, we are requesting all those who are interested in participating to contact Skip Carter at:
skip@taygeta.oc.nps.navy.mil

Those who volunteer to help will be sent a coding guideline and a status report, and will be added to the central mailing list:
scilib@taygeta.oc.nps.navy.mil

which has been established to let participants correspond efficiently.

- Use the board to advertise FIG, and post the table of contents of recent issues of *Forth Dimensions.*
- Allow *anyone* to post and read listings *pertaining to Forth employment.* Don't worry about verifying job postings. The principal goal is to announce *leads*, and not necessarily certified openings. You might, however, have a very few categories for postings, such as definite openings, possible openings, and employment wanted.
- Don't do something foolish like restricting access to FIG members—such a policy will only ensure that *prospective* members never learn enough about FIG to want to join.
- Don't attempt to delete postings which have been filled: bulletin board postings are dated automatically, and out-of-date listings will be obvious. Besides, an out-of-date listing may still provide a useful contact, inasmuch as it indicates an outfit which is using or has used Forth.

I think a free FIG computer bulletin board devoted to Forth job referrals should become rather popular, and should serve to call a great deal of attention to FIG. Once you have an audience, use the opportunity to sell FIG.

Yours truly,
Russell Harris
Houston, Texas

P.S. I know of another Forth opening in Houston. Would FIG be willing to post it? Is anyone out there desperate enough to endure the mosquitoes, the heat, and the humidity? Is there really a Santa Claus?

Thanks, Russell, for your concern and for your letter.

Our one paid, part-time office staffer's job is to take orders and communicate messages to the appropriate people. Our office is the central place for FIG business communication. In the past, we have had many inquiries about jobs and job availability. We have done our best to disseminate that information and it has been successful. We are not staffed to provide job referrals or to do searches for clients, but we have done what we could knowing that this information is important to FIG members.

In the particular case you mentioned, we understood that the employer was anxious to find a Forth programmer so we passed the information by word of mouth to people who we knew were interested or who let us know they were available between the time you first called and when you next called wondering why we hadn't done more. I know we did our part because we have recently received a letter from one of our members thanking us for our help in the referral to the job you mentioned. He said he was one of the "considered," but that the job was then given to a "local Forth person."

If a job has a longer recruiting period, we will attempt to put the notice in Forth Dimensions *as we have often done. To be accurate, we would request that the person doing the looking at least put it on paper or send us a fax. We would put the advertisement in* Forth Dimensions, *usually gratis, since it would benefit our members, but it would still be an advertisement and the responsibility of the advertiser.*

We cannot do more than help with the big problem of employers and employees finding each other. We are only a volunteer organization trying to help, and in this area we cannot be more. When you are dealing with people's "lives and livelihood" there is plenty of room to get into legal trouble.

I guess what I am saying is, in short, we have done the best we can do even though it may not be perfect.

—John Hall
FIG President

Producing Correct Code

Dear Sir:

Congratulations on *Forth Dimensions* XV/5. It was useful, thought provoking, and timely.

I agreed with most of the content of "Forth Development Environments for Real-Time Control." My exception is their choice of Windows as the development and interface environment. Common practice is not a valid reason to select real-time operator interface software. Windows is also continuously being improved, so validating the entire system becomes an ongoing task.

I am particularly interested in the comments by Mike Elola on a syntaxless language. The question of syntax vs. syntaxless languages used to program a microprocessor is part of the overall problem of producing *correct* code to precisely perform the functions required by a control system or application.

From a management perspective, it seems like a good idea to have the compiler check the source code, as one of several tests of correctness. Assuming the cost of a stable, high-performance compiler rarely exceeds one man-month of programmer time, economics favor a strongly typed syntax language where one phase of testing is the source-compile phase.

Without a carefully defined syntax, syntax checking does not benefit the validation of the program other than forcing a common style of programming and compliance with a language standard.

OCCAM for the transputer seems to have sufficient restrictions in the language that side-effects and default actions are drastically reduced. Parameter passing and variable scope are restricted to a carefully defined syntax. My impression of C is that the syntax is a result of single-pass compiler requirements and not restrictions imposed to force correct constructs. My limited knowledge and ability with C make this an unsupported generalization.

Viewing Forth as a syntaxless language implies (correctly, I think) that the programmer is responsible for ensuring correct parameter passing and data typing. Is it more error prone to include all parameters in each procedure call versus maintaining a mental track of the implied parameters on the stack?

Given my generalizations with C, Forth is comparable to C for producing correct code. Given the implied operands of low-level Forth words, Forth may be more

(Continues on page 37.)

HS/Forth, Forth-79

File Format Sleuth

Bradley R. Olson
Grand Rapids, Michigan

Gripe: Forced To Play Detective

As the half-decade epochs of software history go, we live in the Hidden Years. Anybody who uses an application that *almost* does everything they want knows what I mean. Ten years ago, file formats were no mystery. Why, some application manuals straight out told them to you in an appendix.

Not any more! I wanted to perform a simple filtering on some word processing files. My mind told me the program should be easy to piece together in Forth. I'd just look up info on the file format and get to work. But there was no place to look! Plenty of instructions on how to insert disks and use the function keys, but not a stitch of information concerning file format.

We know the file formats exist. The multitude of import and export functions on the latest word processors give witness to that! And maybe, if we were one of The Big Ones we could cut a deal and get the information we need. Or maybe we could shell out the bucks for a (non-refundable) API or SDK kit from the developer—and hope it has the information we seek.

This is a simple but powerful tool for making evident what is clandestine.

The trend is towards hiding—data hidden under objects, file formats hidden under applications, and hardware hidden under an API. And what hiding strives for is good, but when it comes to data, one eventually needs to deal with what *is* instead of how an application presents it.

A Simple Tool

Enter the File Format Sleuth. Nothing grand, having no clever hacks, it just takes a tedious job and does it with welcome clarity.

Simply put, the Sleuth does a dump of any data file. It does so in hex and ASCII. Nothing new, to be sure. Anybody with Norton Utilities or the like has this capability.

What they don't have, what makes this program handy enough to type in, is the format.[1] Not only does the dump also give decimal values and any other format you care to add (an easy process), it displays all of this *simultaneously* in an easy-to-read manner. The ordinal position of every byte and the several translations of its value are displayed *together.* No more hunting between columns of hex and ASCII, no more guessing at decimal equivalents, no more control characters displayed as meaningless dots.

Moreover, the Sleuth has a rudimentary printer interface that gives you a nice print of the dump *with plenty of room for your own comments.* Need more room? It's easy to add. Rather type your comments into a file instead of marking them on a printout? Just type `FILE-IT` before starting the sleuth dump. Then call up your favorite text editor.

Using the Program

To use the program, give it a filename by typing something like `SLEUTH-FILE" Myfile.doc"`. Select output to screen with `SCREEN-IT` or to the printer with `PRINT-IT`. Then execute the main word `FSLEUTH` to see the dump.

You can also dump from a particular point in the file using `FSLEUTH@`, which expects a file position (as a double number) and number of pages to print on the stack (see code comments for details). `0 0 0 FSLEUTH@` would print all the pages beginning at the start of the file. So would `FF 0 0 FSLEUTH@`.

Adding A New Line Format

The program works by reading a given number of bytes from the data file, displaying those bytes in a number of given formats, and continuing to do so until there is some reason to do something else (like an end of page or end of file).

For sake of terminology, the program refers to the printing of one group of bytes read as a *row.* Each row is made up of several *lines.* Each *line* displays the bytes in a different format by printing each byte in a cell. Also, I use the verbs *print* and *display* interchangeably.

Adding a new display format consists of writing a word to display the line, adding that word into the routine `PRINT-ROW`, and adjusting `ROWS/PAGE` and other format values to keep the display tidy.

Example: sample file, command lines, sample output.

```
A Sample WordPerfect 4.2 File
                                            This line is centered.
This is underlined.
This is bold.
                                                                 This is flush right.
This is the last line.


sleuth-init  ok
sleuth-file" wpsamp.wp"  ok
screen-it  ok
fsleuth


|   0|   1|   2|   3|   4|   5|   6|   7|   8|   9|   A|   B|   C|   D|   E|
|   A|    |   S|   a|   m|   p|   l|   e|    |   W|   o|   r|   d|   P|   e|
|  65|  32|  83|  97| 109| 112| 108| 101|  32|  87| 111| 114| 100|  80| 101|
|  41|  20|  53|  61|  6D|  70|  6C|  65|  20|  57|  6F|  72|  64|  50|  65|

|   F|  10|  11|  12|  13|  14|  15|  16|  17|  18|  19|  1A|  1B|  1C|  1D|
|   r|   f|   e|   c|   t|    |   4|   .|   2|    |   F|   i|   l|   e|  ^J|
| 114| 102| 101|  99| 116|  32|  52|  46|  50|  32|  70| 105| 108| 101|  10|
|  72|  66|  65|  63|  74|  20|  34|  2E|  32|  20|  46|  69|  6C|  65|   A|

|  1E|  1F|  20|  21|  22|  23|  24|  25|  26|  27|  28|  29|  2A|  2B|  2C|
|   +|  ^@|   *|  ^_|   +|   T|   h|   i|   s|    |   l|   i|   n|   e|    |
| 195|   0|  42|  31| 195|  84| 104| 105| 115|  32| 108| 105| 110| 101|  32|
|  C3|   0|  2A|  1F|  C3|  54|  68|  69|  73|  20|  6C|  69|  6E|  65|  20|

|  2D|  2E|  2F|  30|  31|  32|  33|  34|  35|  36|  37|  38|  39|  3A|  3B|
|   i|   s|    |   c|   e|   n|   t|   e|   r|   e|   d|   .|   â|  ^J|   ö|
| 105| 115|  32|  99| 101| 110| 116| 101| 114| 101| 100|  46| 131|  10| 148|
|  69|  73|  20|  63|  65|  6E|  74|  65|  72|  65|  64|  2E|  83|   A|  94|

---- Q = Quit.  R = Rewind 1 Screen.  Other Key = Continue ----
```

Vectoring Makes It Easy

As I began to sketch out the utility, I realized that sometimes I'd like to send output to different places. Most Forths make this easy enough to do. The problem is that printers, screens, and files are different creatures. Sure, they all accept input and make it visible in one form or another. But think a minute. When displaying to the screen, it's nice to be able to back up if you missed something. It's also nice to highlight especially important information. On the other hand, when you send information to a printer, one needs to worry about page ends and margins. On top of that, highlighting on a printer works quite differently than on a screen. Then there's file output. The biggest bugaboo here is that files need to be closed after use.

One could handle this with a good dose of `CASE` statements, but it's the hard way out. First, you check where the output is going and branch one of three ways to set it up. Then, every time you reach the end of a page or want to highlight, you can make the decision again. Finally, you can put in another `CASE` statement when the listing is done to decide how to clean up things and close any files.

In classic *Thinking Forth* style, this program makes such decisions once. The words `SCREEN-IT`, `PRINT-IT`, and `FILE-IT` tell the program where to send the data. It does so by re-vectoring key words. Once. After that, you'll find few conditionals regarding the particularia of output media.

Vectoring also saved me from rewriting a lot of code in `.ROW`, which prints out each row. The generic routine `.LINE` can print all the lines, regardless of their format. All `.ROW` does is change the routine that prints each cell, then let `.LINE` handle the details of printing the line.

The method of vectoring execution I've chosen is HSF's word, `DEFER`. The word `TO` takes a CFA from the stack and makes the `DEFER`red word a synonym with the word from which the CFA was obtained. Many dialects differ on the means to do this. Brodie's `DOER MAKE` facility in the appendix of *Thinking Forth* offers good suggestions on

```
\ FILE FORMAT SLEUTH - pretty, multi-format file dump
\ Program (c) 1994. Bradley R. Olson.  ARR.
\ This implementation written in HS/FORTH REV. 5.0

********** USE, CONVENTIONS, AND CREDITS ----------
(((
*** MAIN WORDS ***   ( In Usual Order of Use )
SLEUTH-INIT    sets up initial values for the file dump program.
SLEUTH-FILE"   ( filename" | )  sets name of data file.
SCREEN-IT      sends consequent output to the screen
FILE-IT        sends consequent output to the file SLEUTH.TXT
PRINT-IT       sends consequent output to the printer
FSLEUTH        dumps the entire contents of the data file.
FSLEUTH@       ( D1 N1 ) prints a N1 pages of the data file begin-
               ning at D1 bytes from the beginning of the file.

*** COMMENT CONVENTIONS
\            indicates rest of line is a comment
(((          begins a long comment closed with ")))"
$            address of counted string count byte
c            byte value
w            word value
'c           address of a byte value
'w           address of a word value

*** CREDITS ***
Many thanks to Jim Callahan and Harvard Softworks for their
inspiration and permission regarding these words:
   ((( VAR IS
Print format derived from Jeff Walden's
   "File Formats For Popular PC Software"
   1986.  NY: J. Wiley.
)))

FIND SLEUTH ?((( SLEUTH FORGET-TASK )))
TASK SLEUTH
DECIMAL

********** VARiables, CONSTANTs and PRIMITIVES ----------
27  CONSTANT ESC
179 CONSTANT VERT-SEPARATOR    \  "|" Pretty horizontal bar on PC
12  CONSTANT FORM-FEED
15  VAR CELLS/ROW              \ Bytes displayed per print row
5   VAR SPACES/CELL            \ Width of print cell
3   VAR ROWS/PAGE              \ # print rows per page/screen
0   VAR LEFT-MARGIN            \ Output margin
0   VAR BYTES-READ             \ Bytes this pass.  0 means EOF.
0.  DVAR FILE-POS              \ Pos in file of first byte in row
0   VAR THE-FILE               \ File Handle
0   VAR ROWS-READ              \
1   VAR CUR-PAGE               \ Current Page
0   VAR MAX-PAGES              \ Pages to print
CREATE  THE-BUF     80  ALLOT
CREATE  INFILE$     63  ALLOT
CREATE  OUTFILE$    63  ALLOT

: BYTES/PAGE  ROWS/PAGE CELLS/ROW * ;
: END-OF-PAGE?  ROWS-READ ROWS/PAGE MOD  0= ;


********** MODULE:FILE SYSTEM INTERFACE ----------
: OPEN-INFILE  INFILE$  OPEN-R  IS THE-FILE ;
: CLOSE-INFILE THE-FILE CLOSEH ;
: READ-BUF
     0 0 THE-FILE LSEEK+          \ Save current file position
     IS FILE-POS
     LISTS @ THE-BUF
     CELLS/ROW THE-FILE READH     \ Read into THE-BUF
```

how to do vectoring in many dialects. I have purist friends who shy from it, but in my opinion it's a powerful tool, and worth ferreting out. If your dialect comes with the ability, bless its creators—and use it! At least give it a try. It remedies a lot of Boolean headaches!

Getting It Running

This implementation was done in Harvard Softworks' HS/Forth, which is largely Forth-79 compliant, except that it uses -1 for true, like the '83 standard, which makes some mask operations easier. (HS/Forth users soon will find this code available on the Tools/Toys disk.)

Beyond that, there are four main porting quirks, two that I could have avoided and two that I could not.

First the ones I couldn't have avoided. Access to a host file system is notoriously dialect dependent. The key word I've used here is READH, which I've explained in the code comments. You'll just have to find its counterpart. If you're using an MS-DOS Forth, you'll probably find something similar because it's based on a DOS function call.

I also couldn't avoid specifics for printer and screen output. I've included words to condense the listing and underline portions on the printer. The screen I/O highlights parts of the listing for easy reading. There are ample comments to assist with porting. Perhaps the bigger problem is that dialects differ on how you direct output to the printer. HSF does it with the word PRINT, which changes the operation of EMIT and all consequent output.

Now the things I could have avoided. I could have avoided using my strange comment operators, but I like how they make my code look, and the implementation is interesting. I could also have avoided using HS/Forth's VARs, which are

variables that act like constants. The idea shouldn't be foreign to you—HSF ads have touted them for awhile, and a similar syntax appears in MMS Forth's QUANs. The code should make evident how they work. Fetching is done by the VAR name alone. Storing is done by the infix word IS. I find the syntax helpful and Harvard Softworks'[2] implementation very speedy. If you want to try it out, I've included my own version of IS that works with constants in the same way. But don't try to redefine the single DVAR as a double constant and use the same IS. HSF's IS can do that because its VARs have multiple CFAs. My IS wants a single-width CONSTANT and nothing but. You're on your own to make a double-width one. For the few times it's used, I'd just substitute DVARIABLE, D@, and D!.

If you don't want to play with VARs, just change the VARs to VARIABLEs and put a @ after every occurrence of the name *unless* it's preceded by an IS, in which case get rid of the IS and put a ! after the name.

Making It Better

One could add much to the Sleuth. Under MS-DOS, it would make a fine utility if given a command-line interface and saved as an executable file. The same could be done for any other OS, given an appropriate interface.

The display begs for enhancement. One could display the mnemonic codes for control characters (^J is LF, ^M is CR, ^Z is EOF). I didn't add this feature because it lengthens the display and because the codes are very familiar to me. What would be even more helpful, though much more ambitious, is to allow interactive definition of byte codes or even byte strings. For example, in displaying a file I might discover that the value $86 is used in my word processor to signal the underlining. I

```
        \ For this dialect:
        \ READH ( segment offset n handle -- bytes-read )
        \ tries to read n bytes into the buffer at segment:offset
        \ from the file handle.
        \ It returns actual # of bytes read.
      IS BYTES-READ                   \ 0 bytes read signals EOF
      ROWS-READ 1+ IS ROWS-READ ;

********** MODULE:SCREEN, FILE, PRINTER PRINTER OUTPUT ----------

\ ------- WORDS SHARED IN VARIOUS FORMS OF OUTPUT
: VS  VERT-SEPARATOR EMIT ;     \ Print vertical separator
DEFER .NEW-PAGE                 \ Deal with full page or screen
DEFER .UL+  DEFER .UL-          \ Turn underline on and off
DEFER INIT-OUTPUT               \ Clear or setup output channel
DEFER CLEANUP-OUTPUT            \ Set's output back to normal


\ ------- WORDS FOR SCREEN OUTPUT
: FORCE-QUIT  1 IS MAX-PAGES 2 IS CUR-PAGE ;
: REWIND       \ Try to back up one screen
((( NOTE: remember that on entering REWIND, the DOS file pos
pointer is positioned just past the page we just displayed.  So,
to display the previous page we have to move the pointer back 2
pages! )))
    BYTES/PAGE 2* S->D               \ How far back is 2 scrns?
    DDUP                             \ Hold that thought!
    0 0 THE-FILE LSEEK+              \ Get current position
    D> 0=  ( saves loading D<= )     \ Room to back up 2 screens?
    IF DNEGATE                       \ YES, shift to into reverse
       THE-FILE LSEEK+ DDROP         \ ...and relative disk seek
       ROWS-READ ROWS/PAGE 2* -
       IS ROWS-READ                \ ...and adjust rows read
    ELSE                             \ NO, just go to the beginning
      DDROP
      0 0 THE-FILE LSEEK DDROP
      0 IS ROWS-READ
    THEN ;
: PAUSE/QUIT/REWIND  ( -- f )  \ Waits for a key.
((( Q quits. R rewinds. Anything else continues with next page. )))
   KEY
   BEGIN-CASE
      ASCII Q CASE-OF    FORCE-QUIT  ELSE
      ASCII q CASE-OF    FORCE-QUIT  ELSE
      ASCII R CASE-OF    REWIND      ELSE
      ASCII r CASE-OF    REWIND      ELSE
      ( OTHER CASES )    DROP
   END-CASE ;
: SCR-WAIT
   0 IS CUR-PAGE                     \ Fake out MAX-PAGES check
   CR
   ." --- Q = Quit.  R = Rewind 1 Screen.  Other Key = Continue ---"
   PAUSE/QUIT/REWIND WIPE ;
: INVERSE-SCREEN                     \ Dialect and machine Specific
((( Works in HSF for IBM screens by inverting the least and most
significant bytes of the screen attribute word. )))
   WATRS                \ Get screen attribute word
   DUP [ HEX ] F0 AND   \ Mask out lower byte
   10 /                 \ ... shift msb to lsb
   08 OR                \ and turn on hi-intensity  bit
   SWAP 07 AND          \ Mask out msb and any blink bit (bit 4)
   10 *                 \ and shift lsb to msb
   OR                   \ Put them back together inverted
   WATRS! [ DECIMAL ] ; \ Store as new screen attributes.

DECIMAL
\ ------- WORDS FOR PRINTER OUTPUT
: ISSUE-FF  FORM-FEED EMIT ;
: PRINT-FOOTER
```

```
   CR CR  50 SPACES
   INFILE$ $.  ."    PAGE " DECIMAL CUR-PAGE . ;
: PRINTER-PG  PRINT-FOOTER ISSUE-FF ;

((( The following escape codes are made for my EPSON STYLUS-800,
but should work with most Epson-compatible printers. )))
: PRINTER-UL  ESC EMIT  45 EMIT ;
: PRINTER-UL+ PRINTER-UL 1 EMIT ;
: PRINTER-UL- PRINTER-UL 0 EMIT ;
: PRINTER-INIT
   PRINT                           \ Vector output to printer
   ESC EMIT  ASCII @ EMIT          \ Init Epson Printer
   ESC EMIT  40 EMIT  116 EMIT     \ Standard character set
     3 EMIT 0 EMIT
     0 EMIT 1 EMIT 0 EMIT
     ESC EMIT 116 EMIT 0 EMIT
   15 EMIT   ;                     \ Condensed Print ON

: PRINTER-CLEANUP
   END-OF-PAGE? 0=                 \ Page left in printer?
   IF PRINTER-PG THEN              \ YES, spit it out.
   CRT ;                           \ Turn off printer

\ ------- WORDS FOR FILE OUTPUT
: FILE-INIT
  OUTFILE$ MAKE-OUTPUT             \ Select output file
  >FILE ;                          \ Revector EMIT to that file
: FILE-CLEANUP
   CLOSE-OUTPUT CRT ;        \ End's HSF file revectoring of EMIT

\ ------- WORDS TO SWITCH OUTPUT
: SCREEN-IT    \ sends consequent output to the screen
  CRT                              \ Revector EMIT to screen
  4 IS ROWS/PAGE
  3 IS LEFT-MARGIN
  CFA' SCR-WAIT TO .NEW-PAGE
  CFA' INVERSE-SCREEN TO .UL+
  CFA' INVERSE-SCREEN TO .UL-
  CFA' WIPE TO INIT-OUTPUT         \ Clear screen for starters
  CFA' NOP TO CLEANUP-OUTPUT ;     \ Could reset scr colors here
: FILE-IT      \ sends consequent output to the file SLEUTH.TXT
  0 IS LEFT-MARGIN
  10 IS ROWS/PAGE
  CFA' NOP TO .NEW-PAGE
  CFA' NOP TO .UL+
  CFA' NOP TO .UL-
  CFA' FILE-INIT TO INIT-OUTPUT
  CFA' FILE-CLEANUP TO CLEANUP-OUTPUT ;
: PRINT-IT     \ sends consequent output to the printer
  PRINT                            \ Revector EMIT to printer
  11 IS ROWS/PAGE
  5 IS LEFT-MARGIN
  CFA' PRINTER-PG TO .NEW-PAGE
  CFA' PRINTER-UL+ TO .UL+
  CFA' PRINTER-UL- TO .UL-
  CFA' PRINTER-INIT TO INIT-OUTPUT
  CFA' PRINTER-CLEANUP TO CLEANUP-OUTPUT ;


********** MODULE:FORMAT AND OUTPUT ROWS ----------
((( Terminology:  Each page or screen is composed of n rows.
Each row is composed of n lines.  Each line is composed of n
cells.  Each cell represents a single byte in the data file. )))

\ ------- CELL PRINTING
((( Most of the following are possible vectors to .CELL and
expect the address of a character on the stack. )))

DEFER .CELL  ( 'c )  \ Print a single cell
```

could create a syntax that allows:
`HEX 86 MEANS" UL+"`

...and then subsequent listings would show my symbol, `UL+` whenever it encountered a hex byte 86 in the data file.

If your ambition leads you to some interesting refinements, let me know! As it is, the File Format Sleuth is a simple but powerful tool for making evident what is clandestine. May you find it a good companion in these hidden years.

1. The idea for the display format came from *File Formats for Popular PC Software.* The algorithms and Forth implementation were developed independently.

2. The words for multi-line comments were inspired by Harvard Softworks' code and appear here with permission from and thanks to that company. The implementation of `IS` was inspired by HS/Forth's syntax.

The author relates that he is an ordained minister and novelist who makes a lot of his own tools to work with words. For him, Forth is "...an inspiration, a hobby, and a working tool."

```
: .BLANK-CELL   DROP  VS  SPACES/CELL 1- SPACES ;
: .NUM-CELL  \ Print's cell in any base
     VS C@ SPACES/CELL 1- .R ;
: .CTRL   ( c ) ASCII ^ EMIT  64 +  EMIT ;
: .NORM   ( c ) SPACE EMIT ;
: .ALPHA  ( c ) DUP 32 < IF .CTRL ELSE .NORM THEN ;
: .LETTER-CELL   ( Caddr )
   VS  SPACES/CELL 3 - SPACES C@ .ALPHA ;
: .POSITION-CELL ( 'c )
     DROP
     FILE-POS I' S->D D+
     VS
     SPACES/CELL 1- DU.R ;
((( We ought to use DU.R, since position is always positive, but not all systems have such
a word for fixed-width output of an unsigned double number. If not, the phrase D->S U.R.
will only cause errors in large files. )))

\ ------- LINE AND ROW PRINTING
: .LINE
     LEFT-MARGIN SPACES
     CELLS/ROW 0
     DO I
        DUP BYTES-READ =
        IF CFA' .BLANK-CELL TO .CELL THEN
        THE-BUF + .CELL
     LOOP VS ;
: .ROW  CR
   \ Print the position line
    HEX      CFA' .POSITION-CELL TO .CELL  .LINE CR
   \ Print alpha equivalents
             CFA' .LETTER-CELL TO .CELL
             .UL+ .LINE .UL- CR          \ Accent this line
   \ Print decimal equivalents
    DECIMAL  CFA' .NUM-CELL TO .CELL  .LINE CR
   \ Print hex equivalents
    HEX      CFA' .NUM-CELL TO .CELL  .LINE CR
    END-OF-PAGE?
    IF .NEW-PAGE  CUR-PAGE 1+ IS  CUR-PAGE THEN ;

: READ.ROW      \ Read the next set of bytes and print them.
   READ-BUF
   BYTES-READ 0>
   IF .ROW THEN ;

********** MODULE:OPERATIONAL WORDS ----------

: SLEUTH-RESET-VARS
  0 IS BYTES-READ  0 IS THE-FILE
  0 IS ROWS-READ   1 IS CUR-PAGE ;

: SLEUTH-INIT  \ sets up values for the file dump program.
  $" WPSAMP.WP"  INFILE$  $!        \ Default data file.
  $" SLEUTH.TXT" OUTFILE$ $!        \ Default output file.
  SLEUTH-RESET-VARS  SCREEN-IT ;

: SLEUTH-FILE" \ ( filename" | )  sets name of data file.
    ASCII " WORD                   \ Find next word
    ( $TheWord ) INFILE$ $! ;      \ Save it

: FSLEUTH@  ( D1 N1 )   \ prints certain pages of data
    ((( Begin printing at position D1, and print up to N1 pages )))
    SLEUTH-RESET-VARS
    ( N1 ) ?DUP 0=                 \ 0 pgs becomes MAXINT pages
    IF -1 THEN IS MAX-PAGES        \ MAX-PAGES IS UNSIGNED
    INIT-OUTPUT   OPEN-INFILE      \ Prep i/o  devices
    ( D1 ) THE-FILE LSEEK DDROP    \ Move to desired spot in file
    BEGIN
```

```
      READ.ROW
      BYTES-READ CELLS/ROW <          \ IF No more data left
      CUR-PAGE 1- MAX-PAGES  U< 0= OR  \ OR last page
    UNTIL                                         \ THEN exit loop
    CLOSE-INFILE
    CLEANUP-OUTPUT  SCREEN-IT ;

: FSLEUTH      \ dumps the entire contents of the data file.
   0 0 0 FSLEUTH@ ;

EXIT

********** test drive ----------
SLEUTH-INIT
PRINT-IT
FSLEUTH

********** POSTLUDE - PORTING LONG COMMENTS AND VARS ----------

EXIT  \ None of what follows gets compiled.

\ To do multi-line comments in HS/FORTH, try this:

\ ((( - multi-line comments for HS/FORTH
\ Allows multi-line comments in ((( ... ))) pairs.
\ Just gobbles up the intervening text by parsing
\ through it with WORD.  Algorithm derived from SHOWing
\ HSF's own word ?(((  .  Modified by Brad Olson, 1994.

HEX
: CLOSER?  ( $ -- f )  \ TRUE IF WE'VE FOUND A CLOSE OF COMMENT
   DUP>R  C@          0=         \ Close comment if word is null length
   R@     D@ 29292903. D=  OR     \ ...OR found a ")))"
   R>     D@ 2D2D2D0A. D=  OR ;   \ ...OR found a "----------"
: OPENER?  ( $ -- f )  \ TRUE IF WE'VE FOUND A COMMENT OPENER
   \ Used to flag nested comments as possible errors.
   DUP>R  D@ 28282803. D=         \ Found a "((("
   R>     D@ 2A2A2A0A. D=  OR ;   \ ...OR found a "**********"
: .OPENCMT   CR HERE $. ."  Comment:" ;
: .CLOSECMT  ." .." HERE $. 2 SPACES ;
: EAT-TEXT
  \ As implemented, don't use the same strings for
  \ openers as closers
  BEGIN
    BL AWORD
    DUP OPENER?               \ If the next word opens a comment
    IF  SPACE $.              \ ...print that opener
        BL AWORD
        SPACE DUP $.          \ ...and the following word
        ."  ?? "              \ ...and submit that word to closer?
    THEN
    CLOSER?
  UNTIL .CLOSECMT ;
: (((
   \ Doesn't handle nested comments yet.
   \ Just flags when an opening word appears within a comment
   .OPENCMT
   BL AWORD DUP CLOSER?
   IF   .CLOSECMT
   ELSE $. ." .."               \ Print first word in comment
        EAT-TEXT
   THEN ;  IMMEDIATE
SYNONYM ********** ((( IMMEDIATE

((( PORTING NOTE for long comments
The only dialect-dependent word in "(((" is AWORD.  If WORD in your dialect will search
across end-of-lines, just use WORD.  If not, search your documentation for something that
will. )))
```

```
\ To emulate HSF's VAR's in another FORTH, try this.
((( The resulting words follow HS/FORTH syntax, but HS/FORTH's is twice as efficient. I've
commented heavily to assist porting.  It's written for FORTH-79.  FORTH-83 "tick" acts
different.  >BODY would have to follow the tick, and the IS1 would have to read [COMPILE]
['] )))

: VAR CONSTANT;
: IS1  ( word1 | -- )              \ Compile-time version of IS.
  \ Compiles LIT, the PFA of word1, and the ! command.
     [COMPILE] ' \ during compile, FORTH-79's
                 \ "tick" will compile LIT and the PFA of next
                 \ word in the compile stream
     COMPILE ! ; \ then we compile a store!

: IS0 ( word1 | n -- )             \ Interpret-time version
     [COMPILE] ' \ find PFA of word1
     ! ;         \ and store n there
: IS
  STATE @        \ Are we compiling?
  IF IS1         \ Yes, do a compile-time IS
  ELSE IS0       \ NO, do an interpret-time IS
  THEN ; immediate
```


Total control with *LMI FORTH*™

For Programming Professionals: an expanding family of compatible, high-performance, compilers for microcomputers

For Development:
Interactive Forth-83 Interpreter/Compilers for MS-DOS, 80386 32-bit protected mode, and Microsoft Windows™

- Editor and assembler included
- Uses standard operating system files
- 500 page manual written in plain English
- Support for graphics, floating point, native code generation

For Applications: Forth-83 Metacompiler

- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 64180, 680X0 family, 80X86 family, 80X96/97 family, 8051/31 family, 6303, 6809, 68HC11
- No license fee or royalty for compiled applications

LMI *Laboratory Microsystems Incorporated*
Post Office Box 10430, Marina Del Rey, CA 90295
Phone Credit Card Orders to: (310) 306-7412
Fax: (310) 301-0761

FORTH and *Classic* Computer Support

For that second view on FORTH applications, check out ***The Computer Journal***. If you run an obsolete computer (non-clone or PC/XT clone) and are interested in finding support, then look no further than *TCJ*. We have hardware and software projects, plus support for Kaypros, S100, CP/M, 6809's, PC/XT's, and embedded systems.

Eight bit systems have been our mainstay for TEN years and FORTH is spoken here. We provide printed listings and projects that can run on any system. We provide old fashioned support for older systems. All this for just $24 a year! Get a ***FREE*** sample issue by calling:

(800) 424-8825

TCJ The Computer Journal
PO Box 535
Lincoln, CA 95648

Engineering Notation with Integer Math

Richard W. Fergus
Lombard, Illinois

Why Do We Need This?

Data acquisition applications generally involve accumulating data from transducers, massaging the data, and presenting the results in a readable form. Standard Forth words are very satisfactory for accumulating and massaging data but may not provide an appropriate format for data display. The data display format should account for not only data range but also data precision. In many cases, the precision is much less that the number of digits required for the data range. Displaying digits which are not significant not only clutters the readout but can add confusion when reading the results.

Floating-point math is one solution to this problem but is usually not available on small systems. With Forth "style," engineering notation definitions will be described which can provide engineering notation with selectable significant digits and decimal point positioning while using integer arithmetic. In addition, the exponent notation can be offset to account for data scaling, units, and unit prefixes.

The input number 100. would be represented by .001 or 1.000E-3 in the displayed output.

For example, if a transducer produces an output of 100 integer units for a one milliamperes input and the readout is required to be in amperes, the following illustrates a solution for that application:

```
100, -5 ENG+. --> 1.000E-3 amperes
```

How It Is Done

Four screens have been assembled to describe the engineering notation definitions. Only standard Forth words have been used; therefore, the screens should load without difficulty. Screen One defines the assigns constants for the maximum decimal point positions (number of places to the right of the decimal point) and significant digits. An alternate method of using variables is also described. The variable method allows for easy format modification at any time, but is slower than the constant method. In either case, the decimal point position (`#DP`) must be three or larger. The significant digit range (`LO[` and `]HI`) can be any number of digits. Obviously, many combinations will be not be usable. In general, the significant digits should be equal to or greater than the decimal point position.

Screen Two defines the word which scales the number until it is within the required significant digit range. The `SCALE` word returns a scaled, double-length number and an appropriate exponent value. Two `BEGIN WHILE` loops are used to either multiply (scale up) or divide (scale down) by ten until the number is within the significant digit range. As the number is scaled, the exponent is incremented or decremented to account for the scaling. A scaled number with exponent is returned to the calling word. The scale down has an added feature of rounding off the remainder after the division. A special multiply-by-ten word is defined on Screen One, since double-length multiply words are not available in all Forth systems.

Screen Three defines the basic formatting word. This word requires a double-length number and exponent offset on the stack. First the current number base and exponent are pushed to the return stack. Line four prepares the input number for string conversion, which will be done in two steps. Lines five and six either scale the number or set the exponent if the number is zero.

The exponent offset is pulled from the return stack and added to the exponent passed by the scaling word. A multiple of three power is generated from this sum (the remainder will be used to count the decimal places). The first conversion step that converts the "power of three" is prepared for formatting on line eight. The `<#` initiates a number conversion of the exponent on line nine. After an "E" is added to the string, the remains of the exponent number is dropped from the stack, leaving the sign, scaled number, and the exponent remainder. Line 12 subtracts the remainder of the `/MOD` operation from the maximum number of decimal point positions, and a `DO LOOP` is formed from this value. At this time, the scaled input number is on top of the stack. The `DO LOOP` converts the

```
Screen 1
 0 \                    ENGINEERING FORMAT
 1 FORTH DEFINITIONS DECIMAL
 2
 3 \    *****Constant version--faster
 4   3 CONSTANT #DP   1000, 2CONSTANT LO[   9999, 2CONSTANT ]HI
 5
 6 \    *****Variable version--adaptable
 7 \ VARIABLE (DP    5 (DP !          \ Maximum DP positions
 8 \ : #DP ( --- n ) (DP @ ;          \    minimum of 3
 9 \ 2VARIABLE LO(   100000, LO( 2!   \ Significant digit range
10 \ : LO[ ( --- d ) LO( 2@ ;
11 \ 2VARIABLE )HI   999999, )HI 2!
12 \ : ]HI ( --- d ) )HI 2@ ;
13 : D*10 ( d --- d )
14     2DUP D+ 2DUP                  \ Multiply by 10
15     2DUP D+ 2DUP D+ D+ ; -->      \   in lieu of 10, D*

Screen 2
 0 \                    ENGINEERING FORMAT
 1 FORTH DEFINITIONS DECIMAL
 2 : SCALE ( d --- d n )          \ Scale to required significance
 3     #DP >R                     \ Initial decimal--push to stack
 4     BEGIN                      \ Scale up to lower limit
 5       2DUP LO[ D< WHILE        \ Less than lower significance
 6       R> 1- >R                 \ Move DP to the right
 7       D*10                     \ Multiply by 10
 8     REPEAT
 9     BEGIN                      \ Scale down to upper limit
10       ]HI 2OVER D< WHILE       \ More than higher significance
11       R> 1+ >R                 \ Move DP to the left
12       10  U/MOD                \ Divide by 10
13       ROT 4 > IF 1, D+ THEN    \ If remainder >4, round off
14     REPEAT R> ;                \ Pull exponent off stack
15 -->
```

number of digits to the right of the decimal point. At the conclusion of this loop, a decimal point is added to the string. The remainder of the number is converted by #S and the conversion is closed with #>. Restoring the number base completes the operation.

How to Use It

Several variations of the notation, with or without exponent offset and right justification, are described on Screen Four. There is some freedom in this selection since the exponent offset can be used both to scale the data and to provide unlimited unit selection.

How to select the exponent offset may not be obvious at first. It is simply a matter of counting the decimal place movement to reach the required location (before the engineering notation is applied). From the previous example, the input number (100.) would be represented by .001 or 1.000E-3 in the displayed output. The decimal point was moved five places to the left. Therefore, the exponent offset is a count of the decimal point movement; minus to the left or plus to the right.

This notation has been used for several years in a number of data acquisition applications which involved a wide range of displayed values. Of course, the data was always scaled to be with double significant "Forth" integer range. In some cases, it was possible to increase the dynamic range, with a combination of transducer range change and exponent offset selection, without an apparent change in the format of the data display.

First question—dialects? The screens for the article were written in Uniforth, which is F-83, although the orginal definitions were developed on my own "dialect" (FFORTH).

```
Screen 3
 0 \                      ENGINEERING FORMAT
 1 FORTH DEFINITIONS DECIMAL
 2 : (ENG   ( d n --- adr n )     \ Number/offset to text string
 3     BASE @ >R DECIMAL >R       \ Save base--push offset
 4     2DUP D0< ROT ROT DABS      \ Double number to sign |d|
 5     2DUP OR IF SCALE           \ If not 0, scale between limits
 6       ELSE 0 THEN              \ Leave decimal position for 0
 7     R> + 3 /MOD 3 *            \ Add offset then "powers of 3"
 8     DUP S>D  DABS              \ Exponent to sign |d|
 9     <# #S SIGN                 \ Convert exponent and sign
10       69 HOLD                  \ Place "E"
11       2DROP                    \ Clean stack for next conversion
12       #DP SWAP - 0 DO          \ DP position minus /MOD remainder
13         # LOOP                 \ Do right of decimal point
14       46 HOLD #S SIGN #>       \ Place DP then finish number
15     R> BASE ! ; -->            \ Restore base

Screen 4
 0 \                      OUTPUT VARIATIONS
 1 FORTH DEFINITIONS
 2
 3 : ENG. ( d --- )
 4     0 (ENG TYPE ;                    \ Engineering notation only
 5
 6 : ENG+. ( d off --- )                \ Engineering notation with
 7     (ENG TYPE ;                      \   exponent offset
 8
 9 : ENG+.R ( d off r --- )             \ Above with exponent offset
10     >R (ENG R> OVER - SPACES TYPE ;   \   and right justified
11
12 : ENG.R ( d r --- )                  \ Right justified engineering
13     0 SWAP ENG+.R ;                  \   notation
14
15 ;S
```


The author's experience with Forth began about ten years ago, while developing radiation-monitoring equipment based on the RCA 1802 for a national laboratory. He decided on Forth and, after looking for a while, he finally wrote his own version ("best way to learn Forth!"), and later wrote another for the Motorola 6800. Through the years, he has also used NewMicros MAXForth and Harris RTX packages in a number of applications.

Now retired, Mr. Fergus is heavily involved in a personal severe weather (tornado) warning project which monitors electrical activity from weather fronts. Several Forth-based systems (RCA 1802, Motorola 6800, NewMicros HC11 or HC16, Harris RTX2001, and PC Uniforth) are running continously, collecting and analyzing data.

His development platform consists of PC Uniforth configured as a host for the other Forth packages. He says, "I like the interactive control and limited restrictions of Forth. It allows me to build a program (language) as I see fit. There seems to be a tendency in the current Forth literature to demand an "easier to use language." I like the ability to build an efficient product which might require some "effort" on my part.

Interactive Remote Target Compilation

and the PIC16CXX

Alan M. Robertson
Poole, Dorset, United Kingdom

It may seem, at first sight, an odd choice of processor to run Forth but, as we should all know by now, Forth is applicable to nearly everything—this being especially true of embedded systems. The apparent lack of resources of the Microchip PIC (2K code space, eight-deep stack, and 35 bytes of RAM) need not be a hindrance; indeed, its strength lies in its RISC architecture. In common with other machines, the PIC executes one instruction per machine cycle, the only exception being skip instructions that take two. The other main advantage of the PIC16CXX family is an 18-pin standalone device. The PIC16C84 has another benefit in that the code is E²PROM, so we don't even have to remove to erase.

For Forth to operate with these limited resources, it is necessary to use subroutine threading and to generate native code. The overhead in the PIC for a `CALL-RETURN` is only one word and one machine cycle for each. This, together with in-line code and optimisation, can produce a very compact result.

The Compiler

The host is a PC running a minimal version of F-PC.[1] Onto this is loaded a metacompiler, called `IRTC678`, that performs the native-code generation for the PIC. This compiler runs in two modes, `HOST` and `REMOTE`. The `HOST` mode is really F-PC Forth and performs much as you would expect. This is used to compile code without programming the PIC, for simulating and for generating new compiler directives or additions to the compiler. The `REMOTE` mode makes the PC and target transparent, allowing the PIC to be programmed with compiled code, and for interactive development using the target PIC via the Target Link Monitor (TLM). The programming and the interactive communication with the TLM are both done serially through the ICEPIC hardware. An 8/16-bit switch controls the compiler output. In the eight-bit mode, indicated by the ASCII | character, the data stack is eight bits wide and code is placed in-line and optimised where applicable. In the 16-bit mode, invoked by ASCII | |, the stack is 16 bits wide and routines are compiled that may be `CALL`ed later.

The Library

With the relatively small code space of the PIC16CXX, it is not possible to compile a Forth kernel with all the standard Forth words, and then get in an application. However, most applications do not require all the standard definitions, so if we could only load those necessary, the application space would be much larger. This is the function of the library. The relevant Forth definitions are compiled into the `LIBRARY` vocabulary and executed by the compiler as they are found in the input stream. The definitions in the library are special, but the words `L:` and `L;` allow you to create your own functions to extend the library.

Library words are required to perform two differing functions, depending on their use. If a word is encountered when compiling a colon definition in the target, it is made into what is called a `FORWARD` reference if it has not been previously compiled. When the compiler reaches the end of the current definition, it tests the `TARGET` vocabulary for any such references. These are then looked up in the `LIBRARY` vocabulary to see if they exist. If so, the library definition is executed. This will generate the target code necessary and resolve the references. If, however, the library definition contains other library words, their execution must create a `FORWARD` reference, for now, that will be resolved later. This entire process continues until no more unresolved references exist in the `TARGET` vocabulary that have equivalents in the library. As library words may themselves contain other library words, it is necessary that *no* forward references exist in the library itself. The compiler will give an exception to any word it finds that has not been previously defined.

You may add to the library both assembler and Forth definitions. These are defined as follows:

High level:

```
L: <name> word1 word2 word3 EXIT L;
```

Code:

```
L: <name> M[  MOVLW $55
              DECF FSR
              MOVWF IND0
              RETURN ]M L;
```

Word1, word2, and word3 *must* exist in the library prior to compiling this new high-level definition. M[and]M *must* start and end any code fragments. It is possible to combine both high level and code, but it is necessary to follow the Forth stack rules if undue results are to be avoided. This is possible because we are compiling CALLs to words and then placing the code in-line with them.

Optimisation

As the compiler generates native or machine code, it is possible to further reduce the code size. The PIC has literal equivalents of AND, OR, XOR, ADD, and SUB. If the compiler has just compiled a literal and then is asked to compile an AND, the literal may be replaced by an ANDLW. This is true for eight-bit operation only, but can lead to excellent code generation. Some other areas may also be improved, e.g.,

```
|: ?switch (S - b ) PORTA fC@ 7 AND 1+ ;
```

PORTA and the 1 represent literals, fixed values. fC@, AND, and 1+ are TRANSITION Forth words. In the eight-bit mode, the compiler generates the following code:

```
MOVF PORTA,W
ANDLW 07
DECF FSR
MOVWF (FSR)
INCF (FSR)
RETURN
```

This represents the least amount of code necessary to do the job, full optimisation.

The optimising words are compiled in the TRANSITION vocabulary. Their names are those of Forth: AND, OR, +, -, etc. Their function is dictated by the 8/16-bit switch, and the name added to the TARGET vocabulary will be prefixed by a C or W, depending on the switch. Thus, an eight-bit + may compile in-line code or a C+ routine, whereas a 16-bit + will compile a W+ routine. To improve transparency, a REMOTE vocabulary is placed in the search order when we are in the remote mode. This has the generic Forth words, but allows the appropriate target word to be run interactively, depending on the switch. So, if in remote, we type:

```
| 1 2 + . cr
```

The interpreter will look for the C+ routine. This may not be compiled as a standalone routine, as its use to date may have produced only in-line code. This may be remedied by using REQUIRED C+, which will force the compilation of a C+ routine and header in the target.

Simulation

Often it is necessary to write assembler definitions, even when using a high-level language like Forth. Testing these with the interactive nature of IRTC is generally possible but, in the case of interrupts, this is much more difficult. To assist assembler and Forth programmers, IRTC678 provides a code simulator for the PIC16CXX.

This allows any code fragment to be stepped through, run for n cycles, or run to a breakpoint. The code space may be dumped; the file memory changed; and Forth definitions, interrupts, and PIC registers called by name.

The simulator may be entered with a Forth word and stack values; e.g.,

```
1 2 SIM C+ cr
```

This invokes the simulator with the 1 and 2 placed on the stack in the file memory, and executes the code for an eight-bit +. Pressing the space bar steps the simulator, showing the following:

```
$0188 [C+]
      $0800    MOVF INDO,W
      [WREG=$02       f2D=$02 SP=1]
$0189 $0A84    INCF FSR,f
      [WREG=$02       f04=$2E SP=1]
$018A $0780    ADDWF INDO,f
      [WREG=$02       f2E=$03 SP=1]
$108B $0002    RETURN
      [WREG=$02               SP=0]
```

The WREG is popped with the top stack item, 2, and then the stack pointer, FSR, is incremented to point at the next item, 1. The two are added directly to the stack via INDO.

This tool gives greater confidence of proper code execution prior to interactive testing. The only areas not checked are the operation of on-board hardware like timers. However, the effects of these may be simulated by appropriately setting the file memory contents.

The simulator steps through code compiled into a 64K segment in the host memory. An Intel Hex file utility may save the contents of this code space or load the space from a Hex file. This allows the simulator also to be used on imported code.

Target Link Monitor

The TLM is the communication program that resides in the target PIC to enable execution of compiled code, and inspection and modification of PIC resources. This is all carried out by the word SERVER which is defined as follows:

```
: SERVER ( S - )
  BEGIN >STACK >STACK EXECUTE
  AGAIN ;
```

We must thank the University of Rostock,[3] in what was East Germany, for this deceptively simple idea. The words >STACK wait for the host to send two bytes of data, LSB first, to the target stack. This 16-bit value must be a valid CFA (code field address) of an existing target definition. The word EXECUTE performs a computed GOTO to this

address.

The basic TLM words are:

```
STACK>       >STACK
fC@          SP@
EXECUTE      FREEZE
PIC.RESET
```

With only these, it is possible for `SERVER` to interrogate the PIC internals. For example, if we wish to look at the current value of the status register, we need to perform an `fC@` on file address 03 and send the result to the host for display. This is what happens if we type:

```
STATUS fC@ H. cr
```

Host sends CFA of `>STACK`, `EXECUTE` runs it.
Host sends 03 (`STATUS` adds) which `>STACK` puts on target stack.
Host sends CFA of `fC@`, `EXECUTE` runs it, and `fC@` fetches value to stack.
Host sends CFA of `STACK>`, `EXECUTE` runs it and sends the top-of-stack to host.
Host displays the eight-bit value in Hex.

With the addition of `fC!`, by using `REQUIRED fC!`, the host may modify PIC resources. This allows you to write host programs that require no further target code, but that will exercise hardware external to the PIC. This is very useful in development and production, to do initial circuit testing. The only limitation is the use of the CLK and SDA pins, PortB bits 6 and seven.

Smart-ICEPIC

The Smart-ICEPIC module allows the compiler to program the PIC16C71/84 directly and, via a cable, the PIC16C64 or larger external devices. The ICEPIC has all the necessary logic to provide the +12V for programming and the signals for serial communication on RB6&7. These programming pins are switched by relays to the target hardware, allowing all the PIC port pins to be used during debugging and testing. Also, to facilitate prototyping, an on-board oscillator with nine frequencies from 19.66 MHz to 74.6 KHz is provided.

Conclusions

Yet again, Forth comes to the aid of the developer wishing to make use of minimum resources. Its interactivity gives us a quick and reliable method of solving our application problems. Why should anyone wish to use that What's-C-called language? It beats me.

References

1. F-PC v. 3.5, fat Forth for the PC by Tom Zimmer.
2. *Interactive Remote Compilation for Development and Machine Integration.* Alan M. Robertson, EuroFORML '89.
3. *A Distributed Forth Environment.* Dr. Egmont Woitzel, EuroFORML '90.
4. *The inter-Application Execution of Forth Words for Seamless Cooperative Systems.* Roy Goddard, EuroFORML '90.
5. *A 448 Byte Forth Multitasking Kernel.* Alan M. Robertson, EuroFORML '92.

Copyright © 1994 by RAM Technology Systems, Ltd.

Alan M. Robertson has an honorary degree in electronics from the University of Salford in the United Kingdom. He worked for many years on the design of hardware-based controllers for machine tools, coming to Forth in the late 70's when he started using micros. He has been using Forth ever since. He works as a consultant for industrial embedded control applications for RAM Technology Systems, Ltd., which he founded in 1983.

MAKE YOUR SMALL COMPUTER
THINK BIG
(We've been doing it since 1977 for IBM PC, XT, AT, PS2, and TRS-80 models 1, 3, 4 & 4P.)

FOR THE OFFICE — Simplify and speed your work with our outstanding word processing, database handlers, and general ledger software. They are easy to use, powerful, with executive-look print-outs, reasonable site license costs and comfortable, reliable support. Ralph K. Andrist, author/historian, says: "**FORTHWRITE** lets me concentrate on my manuscript, not the computer." Stewart Johnson, Boston Mailing Co., says: "We use **DATAHANDLER-PLUS** because it's the best we've seen."

MMSFORTH System Disk from $179.95
Modular pricing — Integrate with System Disk only what you need:
FORTHWRITE - Wordprocessor $99.95
DATAHANDLER - Database $59.95
DATAHANDLER-PLUS - Database $99.95
FORTHCOM - for Communications $49.95
GENERAL LEDGER - Accounting System $250.00

MMSFORTH
MILLER MICROCOMPUTER SERVICES
61 Lake Shore Road, Natick, MA 01760
(508/653-6136, 9 am - 9 pm)

FOR PROGRAMMERS — Build programs FASTER and SMALLER with our "Intelligent" MMSFORTH System and applications modules, plus the famous MMSFORTH continuing support. Most modules include source code. Ferren MacIntyre, oceanographer, says: "Forth is the language that microcomputers were invented to run."

SOFTWARE MANUFACTURERS — Efficient software tools save time and money. MMSFORTH's flexibility, compactness and speed have resulted in better products in less time for a wide range of software developers including Ashton-Tate, Excalibur Technologies, Lindbergh Systems, Lockheed Missile and Space Division, and NASA-Goddard.

MMSFORTH V2.4 System Disk from $179.95
Needs only 24K RAM compared to 100K for BASIC, C, Pascal and others. Convert your computer into a Forth virtual machine with sophisticated Forth editor and related tools. This can result in 4 to 10 times greater productivity.

Modular pricing — Integrate with System Disk only what you need:
EXPERT-2 - Expert System Development $69.95
FORTHCOM - Flexible data transfer $49.95
UTILITIES - Graphics, 8087 support and other facilities.

and a little more!

THIRTY-DAY FREE OFFER — **Free MMSFORTH GAMES DISK** worth $39.95, with purchase of MMSFORTH System. CRYPTOQUOTE HELPER, OTHELLO, BREAKFORTH and others.

Call for free brochure, technical info or pricing details.

Switch in Forth

Walter J. Rottenkolber
Mariposa, California

Most Forthwrights are familiar with CASE to replace multiple branching statements. Its C equivalent, Switch, is less commonly seen. So, for frustrated C mavens, the incurably curious, and the enlightened who are converting C to Forth, I present Switch in Forth.

An example of the switch statement:

```
switch ( exp ) {
case  1: one;
default: three;
case  2: two;
         break;
case  3: three;
case  4: four;
}
```

"Exp" can be any expression that produces a value to be matched with the values after "case." If the two values are equal, a match occurs. Then all the statements after the colons are run until either a "break" or the end of switch is reached, including the one after "default." If no match is made, control jumps to "default" as though the match had been made there.

Switch perfectly matches the arcane and inscrutable syntax of the C language.

In this example, if n=1, then statements "one," "three," and "two" would run, with "break" stopping further progress. If n=3, then "three" and "four" would run. N=6 would run the "default" code "three" and "two."

To avoid conflicts with pre-existing words in my Forth, I've modified the C syntax to protect the innocent. In the Forth Switch, the example would appear as:

```
switch: ( n )
case' 1 =:: one ;;
default:   three ;;
case' 2 =:: two  break;
case' 3 =:: three ;;
case' 4 =:: four ;;
endswitch <default ;
```

The value "n" is on the stack. Break; substitutes for ;; where it occurs. Do *not* place it as a separate statement as you would in C. Endswitch must always end the switch statement, but <default is present (after endswitch) only when a default: statement is used. Study the listing for more examples.

The switch code uses the Forth branching words. A flag, DEFLG, marks if a match has occurred. It's a reverse flag, in that a match = false. Originally, DEFLG was in a variable, but I decided that this could be a source of conflict in multitasking systems, as it would not be re-entrant. So I moved it inline within SWITCH: just after JMPWORD. The variable is now used only while compiling the switch statement to hold DEFLG's address.

In the example, the branching goes as shown in Figure One. It uses the fact that the code between IF and THEN can leave a flag to be used by a subsequent IF. Because of the 2SWAP in (::), ?>RESOLVE (the code for THEN) refers to the previous branch, the IF in switch: or the ELSE's, not the immediate IF for the match comparison. As a result, a no-

Figure One.

```
( n)   JMPWORD [DEFLG]   DEFLG ON   TRUE IF
DUP n' = IF-DROP-DEFLG-OFF-THEN   <code>  ELSE
ELSE BEGIN                        <code>  ELSE
DUP n' = IF-DROP-DEFLG-OFF-THEN   <code>  EXIT ELSE
DUP n' = IF-DROP-DEFLG-OFF-THEN   <code>  ELSE
DUP n' = IF-DROP-DEFLG-OFF-THEN   <code>  ELSE
DROP THEN  DEFTOG UNTIL ;
```

To avoid conflicts with pre-existing words in my Forth, I've modified the C syntax to protect the innocent.

match condition jumps to the next comparison, whereas once the <code> runs, the jump is to the next <code>.

`Default` is a `BEGIN UNTIL` loop that uses `DEFLG`. `DEFLG` is originally set to true. If a match occurs, `DEFLG` is set false so that when it is toggled in `DEFTOG` it becomes true and `default` is by-passed. If no match occurs, then `DEFLG` toggles false, and the back branch occurs. The next time through `DEFTOG`, `DEFLG` is toggled true and `switch` is exited.

Note that `Break;` exits the `switch` word, so be careful of adding further code between `Endswitch` or `<Default` and `;`, or you may be surprised.

Forth always does more, so I've added the words `<::`, `>::`, and `range::`. These allow tests for "n" values less-than, greater-than, and range-between the selector values after `case'`.

`Switch` perfectly matches the arcane and inscrutable syntax of the C language. Its complexity encourages programming tricks not allowed by the plain logic of `CASE`. `Switch` in Forth follows the logic of the original, permitting nearly direct translation of code from C, including the tricks.

Walter J. Rottenkolber bought his first computer in 1983. Early on, he experimented with fig-Forth and other languages, but gravitated to assembler until re-introduced to Forth in 1988. He notes that Forth provides the same close-to-the-silicon feeling as assembler, but without the pain. Interests include small embedded systems, programming, and computer history, about which he enjoys writing.

```
     1
 0 \ Switch in Forth
 1 VARIABLE DEFLG
 2 : JMPWORD   R> 2+ >R ;
 3 : >DEFLG    HERE DEFLG ! 0 , ; IMMEDIATE
 4 : DEFON   DEFLG @  [COMPILE] LITERAL  COMPILE ON ; IMMEDIATE
 5 : DEFOFF  DEFLG @  [COMPILE] LITERAL  COMPILE OFF ; IMMEDIATE
 6 : SWITCH:  ( n)  COMPILE JMPWORD  [COMPILE] >DEFLG
 7     [COMPILE] DEFON  COMPILE TRUE  [COMPILE] IF ; IMMEDIATE
 8 : CASE'  ( n - n n)  COMPILE DUP ; IMMEDIATE
 9 : (::)  ( n - |n)  [COMPILE] IF 2SWAP  COMPILE DROP
10     [COMPILE] DEFOFF  ?>RESOLVE ; IMMEDIATE
11 : =::  ( n n n' - |n)  COMPILE =  [COMPILE] (::) ; IMMEDIATE
12 : <::  ( n n n' - |n)  COMPILE <  [COMPILE] (::) ; IMMEDIATE
13 : >::  ( n n n' - |n)  COMPILE >  [COMPILE] (::) ; IMMEDIATE
14 \S
15

     2
 0 \ Switch in Forth
 1 : RANGE::  ( n n n' n' - |n)
 2     COMPILE BETWEEN  [COMPILE] (::) ; IMMEDIATE
 3 : ;;   [COMPILE] ELSE ; IMMEDIATE
 4 : BREAK;   COMPILE EXIT  [COMPILE] ELSE ; IMMEDIATE
 5 : DEFAULT:   [COMPILE] ELSE  ?<MARK 2SWAP ; IMMEDIATE
 6 : DEFTOG  ( a - f)  DUP @ 0= TUCK SWAP ! ;
 7 : <DEFAULT   DEFLG @ [COMPILE] LITERAL  COMPILE DEFTOG
 8     [COMPILE] UNTIL ; IMMEDIATE
 9 : ENDSWITCH  ( n)  COMPILE DROP ?>RESOLVE ; IMMEDIATE
10 \S
11
12
13
14
15

     3
 \ Switch in Forth

 : ti$   cr ." This is " ;
 : one    ti$ ." one." ;
 : two    ti$ ." two." ;
 : three  ti$ ." three." ;
 : four   ti$ ." four." ;

     4
 \ Switch in Forth
 : s1
     switch:
     case' 3 =:: three ;;
     case' 4 >:: four break;
     case' 1 =:: one ;;
     case' 2 =:: two ;;
     endswitch ;

 : s2
     switch:
     case' 5 10 RANGE:: three ;;
     case' 4 =:: four ;;
     default:   one ;;
     case' 2 <:: two ;;
     endswitch <default ;
```


ADVERTISERS INDEX

The Computer Journal 14

Forth Interest Group .... centerfold, 40

Laboratory Microsystems, Inc. 14

Miller Microcomputer Services 20

Silicon Composers 2

The Essence of Forth...

...is the Relationship Between Programmer and Source

Randy Leberknight and Dennis Ruffer
FORTH, Inc., Manhattan Beach, California

It has long been observed that Forth has an almost magical effect on the productivity of a programmer. Chuck Moore's original aim in developing Forth was to increase his productivity, and he has estimated the increase was more than ten-fold. Many of us have real-life experiences with projects estimated at <n> months in C (or whatever) that were completed in <n> *weeks* or less using Forth. Thoughtful observers of this phenomenon agree that the magic lies in the intimate relationship between the programmer and the program under development.

Recently, we at FORTH, Inc. have been studying platforms and programming environments in order to determine how our systems of the future should look and act. We've been using various programming tools, including Forths on other platforms such as Windows and UNIX. The major deficiency we have found with other approaches is that they seem procedurally to separate the programmer from the source more than is comfortable.

For us as programmers, everything we do has some connection with source code. Whether editing, compiling, or debugging, we are performing these functions on source code. While editing, we make the source code look right; while compiling, we make it fit together in an executable fashion; while debugging, we make it work as intended. All this time, it is only the source code and how it expresses the application that really matters.

Intimacy with the source code is being lost and, with it, much of the "magic."

The traditional (i.e., TTY) Forth interface is an interactive, line-oriented command interface. From the original dozen or so editing commands that the very early (and very resource-constrained) Forths supported, we have expanded our set of editing commands in our "traditional" editor to over 70 (most of which are single-keystroke or function key commands), plus rich command sets for managing source and related resources. In addition, we offer two "full-screen" editors (one incorporated in our GUI toolkit) and an optional text-file-management facility. However, most of us find the extended command-line interface to yield the most intimate—and most productive—relationship with the source.

Recently, we had some experience with a large Forth application in a UNIX environment. We have had little exposure to this world, and were more than a little curious about what is often promoted as the ultimate programming environment for the black-belt programmers with turbo-props on their beanies. The users had networked workstations, their favorite editors, source-management utilities, grep, make files, etc... This system is very large, and there are literally thousands of source files and hundreds of directories. Due to issues such as version management and the size of the application, we could see they really needed all those directories and files, and the UNIX network seemed to be a good place to keep them.

But in exploring this complex system with their experts, we were appalled at just how difficult it was to perform simple actions that we had come to take for granted. For example, to find the source for a particular word was a nightmare. We had multiple editor sessions going, including one with a cross-reference list showing the path and file name of the source file for each word. To find our word, we had to:

1. Leave the current window and switch to the file with the cross-reference list.
2. Search the cross-reference file for an occurrence of the target text string (sometimes skipping matching strings which were substrings of larger words).
3. Note the file name containing the source for the word in question (including a path which could be four or five levels deep, requiring accurate typing in subsequent steps).
4. Begin an editing session on that file.
5. Do another text search on this file to find the actual definition of the word.
6. Try to remember why we wanted to find it in the first place.

Back home, we would simply have typed:

```
LOCATE <wordname>
```

"Help screen" for polyFORTH GUI screen editor. In addition to these commands, `PgUp` and `PgDn` move forward and back one block, respectively, and buttons are available for common functions.

```
Cursor keys :                             Special Keys:
Arrow keys move cursor                     Ctl-F  .... Search for text
Ctl-Right Arrow . Right one word           Ctl-R  .... Replace found text
Ctl-Left Arrow .. Left one word            Ctl-O  .... Edit last selected block
Home ........... Start of line             Insert .... Toggles Insert mode
End  ........... End of line
Ctl-Home  ....... Top of screen            In Wrap mode :
Ctl-End  ........ End of screen               Ctl-Enter ... Justifies to next line
Ctl-PgUp  ....... Go to start of file         Ctl-Del ..... Deletes spaces
Ctl-PgDn  ....... Go to end of file        In Clip mode :
                                              Ctl-Enter ... Inserts new line
                                              Ctl-Del ..... Deletes line if blank
Working with Selected text :
  Holding down the shift key while moving the cursor selects text.

  Del ..... Deletes selected text
  Ctl-C ... Copies selected text to clipboard
  Ctl-V ... Inserts text from clipboard
  Ctl-X ... Copies selected text to clipboard then deletes it
  Insertions replace selected text.    Double clicking selects a word.

Other Editing Keys :
F1  ........ Toggle screen editor     F6  ........ Locate compiled word
F2  ........ Toggle Clip/Wrap         F7  ........ Edit shadow block
F3  ........ Save to disk             F8  ........ Revert block to last saved
F4  ........ Cross-reference word     F9  ........ Acknowledge alarms
F5  ........ Recompile application    F10 ........ Display this help screen
```

to get the source displayed immediately.

We have also experimented with a Windows Forth that provides an interactive window attached to your application, with some debugging tools. But there is no editing capability in that window, and no access to source. Editing is done in a completely separate window, either with an editor provided with the system or your favorite commercial editor. As in the UNIX case, the lack of a direct connection leads to clumsy procedures that break your concentration just when it's most important.

There's a lot more to using polyFORTH than just being able to type `DUP DROP` and `LOAD`. There is being able to type `L` after an error occurs and seeing the offending source, with the problematic text highlighted. There is being able to fix the problem right there and continue compiling immediately. There is using `LOCATE` to let us see a word we don't remember, and:

```
WH <wordname>
```

to get a cross-reference of WHere it's used. Having `LOCATE`d a word, `Q` shows us its documentation, and `O` returns to the point in the source we were before the `LOCATE`. (the Other block). There is the string editor that lets us type things like:

```
F DUP^ R DROP^ MANY
```

the effect of which is to find the string `DUP`, replace it with `DROP`, and repeatedly interpret this command line until no more occurrences are found. Then there are the tools that make the management of a modular, hierarchical source organization almost effortless: `QX`, which displays a "quick index" of the first (comment) lines of each of a set of 60 blocks; its siblings `NX`, `LX`, and `BX` that let you move around in this "index space"; and a similar set of file-management words. The figures show "help screens" for the GUI screen editor and enhanced character editor.

In our polyFORTH environment, blocks and groups of blocks provide what seems to us a more convenient and manageable level of modularity than the files, and multiple host OS files occupy the role of directories in the UNIX or Windows environments. We may still have thousands of blocks and hundreds of files open concurrently for a complex application, but functions like `LOCATE` and `WH` work instantaneously, making the entire environment instantly within reach.

Although these tools are optimized for a block-oriented source system, which we find to be most convenient to use, we have versions of most of them that connect to our

"Help screen" for polyFORTH string editor. In addition to these commands, `PgUp` and `PgDn` move forward and back one block, respectively, and function-key equivalents are available for common functions.

```
See the polyFORTH Reference Manual for more on String Editor commands.

N  List next block                 F  Find specified text, keeping it in Find buffer
B  List previous block             E  Delete the last found text
Q  List shadow block               I  Insert string, keeping it in Insert buffer
O  List other block                K  Swap find and insert buffers
T  Move to line number             M  Move lines from another block
P  Put string at current line      L  Relist the current block
U  Put string under current line   A  Adjust text (delete to 1st non-blank)
X  Delete current line             J  Justify text (next non-blank starts new line)
D  Delete specified text           S  Search blocks for a string

Arrows  Move cursor                PART    Change to another file part
RETAIN  Copy line to insert buffer LOAD    Load the block number
TILL    Delete till specified text FINISH  Load the rest of the block
UNDO    Revert block to last saved WH      Cross-reference a word
FLUSH   Save blocks to disk        TABS    Display tabs
LX      Show current 60-blk index  CUT     Cut lines
QX      Show specified index       PASTE   Paste lines
NX, BX  Show Next, prev. indexes   .S      Display stack
CLIP    Change to clip mode        YANK    Copy lines
WRAP    Change to wrap mode        IN      Block number of specified file
COPY    Copy block                 BEYOND  Last block number in source code
LIST    Edit specified block       LOCATE  Locate a word

MULTIPLE  Reinterpret the current command line with query
MANY      Reinterpret the current command line
```

optional text-file interpreter as well. The issue isn't "blocks vs. files," it's the programmer's relationship to the source.

In the absence of easy access to source, most systems rely on decompilers and disassemblers. We have a decompiler and disassembler, but generally would much prefer to work with source. When you go to the source, you see a lot more than the definition of the word. You see its stack comment, the context of related words in which it was defined, any special compiling techniques used in its construction, and with a single keystroke (`Q`) can access its documentation. Any of this may be crucial information. Most importantly, if you now wish to make a change, you're *there*, all ready to go, with no further procedure required. A decompiler shows you different kinds of information: the effects of vectoring, compile-time actions, pointers to run-time code, re-vectoring, etc. The decompiler shows you *what* happened in the compiling process; the source shows *how* it happened, and provides the mechanism for changing it. Both can be important; the latter is the mainstream of programming life.

Forth has traditionally been thought of as an integrated editor, compiler, and linker, but as we think about how we actually use it, we see that the editor is really where we spend our lives, and everything else can be seen as merely extensions of the editing environment. The editor needs to be intimately connected to the running application, so we can find definitions easily (e.g., `LOCATE`), and able to index words as it encounters them (for `WH`). It needs a command-type interface, so we can flexibly combine repetitive commands into application-specific extensions. And it needs a way for the system to interact with us so we can tell it to execute a word and it can tell us what it did. All told, this begins to look a lot closer to a special kind of *word processor!*

As we work toward the "dream system" of the 90's, we are working with push-buttons, pull-down menus, icons, and other modern devices. But as systems increasingly compartmentalize these functions, we find that the level of intimacy with the source code that we are accustomed to is being lost, and with it much of the "magic." Using our current tools, we can move anywhere our thoughts take us so fast we never break our train of thought. Zoom out for a broad look, dive into a word, take a side trip, and jump back without thinking about it at all, never more than one command away from an interactive interpreter. That's the magic. The challenge is to use modern tools to enhance it.

Simple Mouse and Button Words

for DOS-Based Systems

Richard C. Wagner
Revere, Pennsylvania

One of the utilities missing from most MS-DOS Forth systems is the now-ubiquitous, point-and-click graphical user interface. This type of environment, which at first may seem like a complex software system, isn't, and can be supplied with a few simple Forth words, comprising only ten blocks of code.

Introduction

One of the greatest detriments Forth programmers using DOS-based systems see in their language is the "primitive" command-line interface. Exposure to commercial applications with point-and-click graphical user interfaces (GUI's) has led programmers to question the power and acceptability of their own applications, and wish they had a Mac- or Windows-based system providing all of the words needed to develop a custom GUI. It seems as though many Forth programmers, out of their desire to have these tools, and not realizing how simple this type of system can be, have resorted to using other languages such as C or (gasp) C++. Let us not forget that, even though the libraries of GUI routines for these languages may appear to be magically available, the guys at Borland and Microsoft had to write the routines using either the high-level language or some form of assembler. Therefore, the concepts and architecture of the system must be straightforward—doable. Like my old Forth mentor and programming partner used to say, "Somebody's already done it, how hard could it be?" And since we're using Forth, we'll probably find it easier than they did.

Graphical user interfaces have led programmers to question the power and acceptability of their own applications...

The GUI development system presented here provides two major sets of words: one set to communicate with the MS-DOS mouse drivers; and another set to display the buttons, detect a button "press," and execute the code associated with a button. This system was written for polyFORTH, using F-83. However, it should work on other Forths, providing the graphics words are replaced with those of the other system, and the `CODE` words be rewritten to work in the other system (there are only eight `CODE` words, so don't sweat it.)

Mouse Words

Communicating with the MS-DOS mouse drivers is straightforward, using `CODE` words to call various functions attached to interrupt 33H. The first order of business is initializing the mouse. `IMOUSE` (on block 101) returns both a true or false flag, depending on the presence or absence of the MS-DOS mouse drivers, and the number of buttons on the mouse. In addition, the mouse position is reset to the middle of the screen, the cursor is turned off, and the default cursor shape (the arrow) is selected (more on this later). Displaying the cursor is done using `.MOUSE`. This causes the drivers to display the cursor at the current mouse location continuously until `-MOUSE` is executed. `MOUSE?` returns the current mouse location, and a number from zero to seven representing the mouse buttons as three bits, least significant to the right. So if this word is executed while the left button is pressed, a four will be returned. `MOUSEAT` ("mouse at") accepts an X and Y screen location, and sets the mouse there. `MOUSE?` and `MOUSEAT` both operate whether or not the cursor is displayed.

The last two major mouse words are `?PRESS` and `?RELEASE` on block 102. These both accept a button number (zero for left; two for center; one for right) and return the number of times that button was pressed or released since the last time this word was executed. They also return the location of the *latest* press or release. These words are very important, for they allow you to determine where the last click was without having to poll the mouse to capture it. Thus, if a button is clicked while your software is off doing something, the event will still be detectable when execution returns to the GUI. The Microsoft Flight Simulator, just for example, ignores this technique, polling the mouse for input. Consequently, a user sometimes must click on a button numerous times before his click and the software's polling cycle coincide, and the system finally responds. See, there's no magic in software

Table One. polyFORTH register assignments.

Intel Name	polyFORTH Name	polyFORTH Use
AX	0	Scratch
CX	1	Scratch (counter)
DX	2	Scratch
BX	U	Base address for user variables
SP	S	Parameter stack pointer
BP	R	Return stack pointer
SI	I	Interpreter pointer
DI	W	CFA pointer, need not be preserved

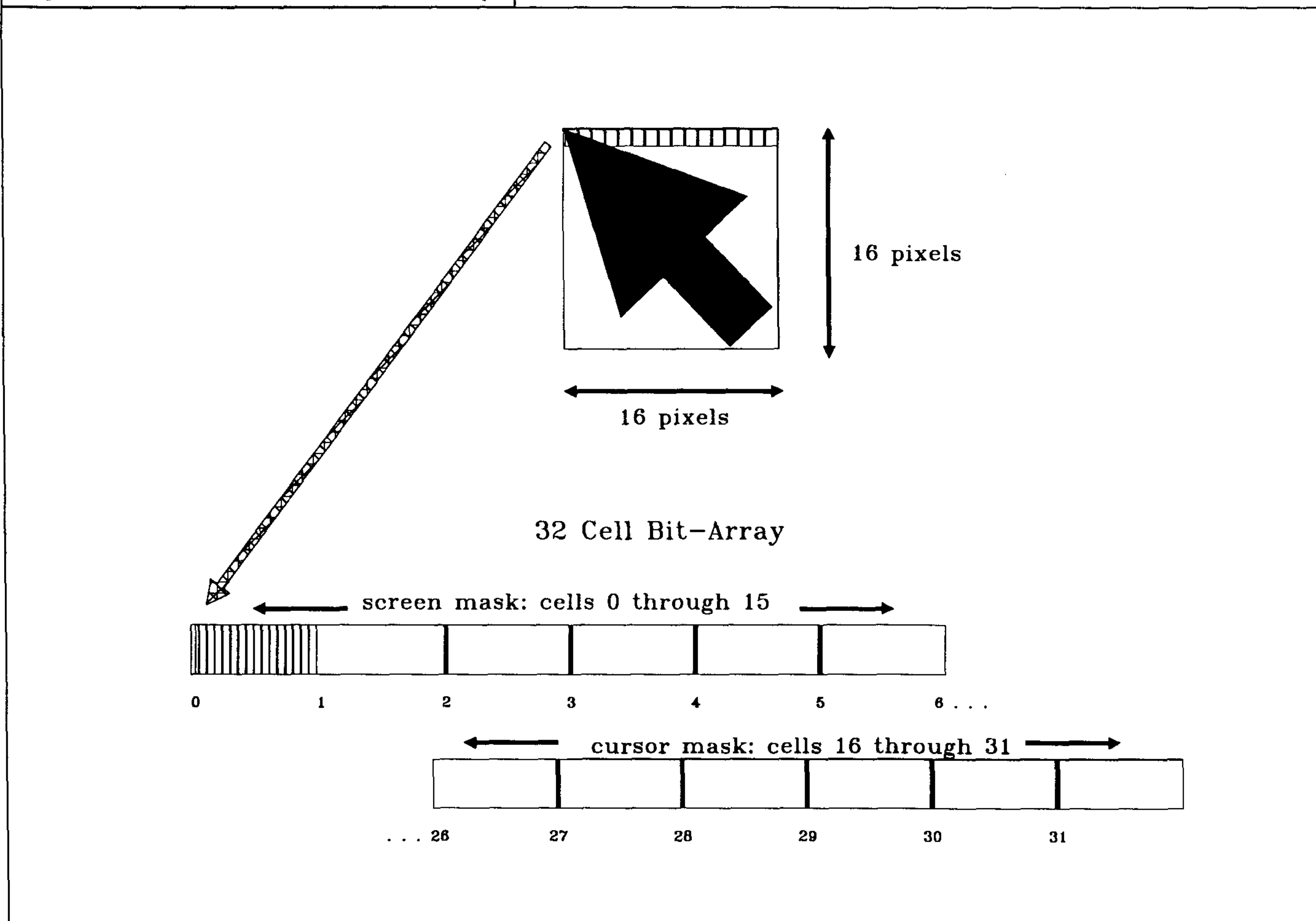
land, and those Microsoft guys aren't smarter than us after all. Worse, they released this system fully aware of the deficient performance.

All of these words work either in text or graphics mode. So for those who don't have a Forth system with graphics words, they can use these words to build a text-based point-and-click interface. Our "UI," however, contains a "G" and, therefore, we'll assume from now on that we're in graphics mode. Should you need to rewrite the `CODE` words for another system, Reference One provides an adequate description of the interrupt calls. The polyFORTH register assignments in Table One will help to reverse-engineer my code.

The last (and optional) mouse words are used to change the shape of the graphics cursor. In most systems employing a GUI, access to the computer is limited strictly to the mouse. After clicking on a button requiring a lot of processing, the user will find the machine unresponsive until it returns to the GUI. It would be nice to let the user know that the machine is busy, using a clock (á la Macintosh) or an hourglass (á la Windows) or (how about this) an "ok" (á la Forth). The DOS mouse drivers provide for this by using a software-defined shape for displaying the cursor.

When the mouse drivers are loaded into the computer, a 32-cell (64-byte) array is established somewhere in memory (you don't have to know where). The data in the array defines the shape of the default arrow cursor displayed by `.MOUSE`. Those kindly Microsoft guys left us with the option of redirecting the mouse drivers to use an alternate cursor-defining array. Changing the shape of the cursor is merely a matter of setting up a new array and

Figure One. The cursor-definition array.



telling the DOS drivers where it is.

As illustrated in Figure One, the graphics cursor is a floating region of display, 16 pixels high by 16 pixels wide. Each bit in the cursor-definition array represents one pixel of the cursor, starting at the upper left. Thus, the zeroth cell of the array represents the top row of the cursor, the number one cell represents the second row, and so on. The attentive reader will note that there are twice as many cells as there are rows. The cursor definition array is actually split into two 16-cell "masks." Cells zero through 15 represent the "screen mask." This mask determines where the cursor will appear transparent or opaque. For transparent cursor pixels, the corresponding bits of the screen mask must be set to "one." The bits corresponding to the opaque cursor pixels must be set to "zero." Cells 16 through 31 of the array contain the "cursor mask." This determines the color of the cursor's opaque pixels. A "zero" bit in this mask, provided it falls in an opaque section of the cursor, yields a black pixel; a "one" bit produces a white pixel. In the transparent portions of the cursor, it doesn't matter how the cursor mask bits are set.

`SET-SHAPE`, on block 104, redirects the DOS mouse drivers to use a new cursor array. It accepts the array's address and the location, within the 16-by-16 pixel cursor field, of the hot spot. This is the actual pixel representing the cursor's position on the graphics screen. For the default arrow cursor, it is set to 0,0 (upper left). Block 103 contains the words which load the masks into the array. `MASK-DEST` is a variable used to temporarily hold the array address and keep it off of the stack. `!ROW` accepts 16 zeroes or ones (along with the cursor row number they represent, and the address of the array) off the stack and packs them into a single 16-bit value. It then stores the value into the correct cell of the array. `!SCREEN-MASK` and `!CURSOR-MASK` execute this word 16 times to fully load each mask into the array. They each accept 256 zeroes or ones off the stack, with the address of the array on top. If your system doesn't have this much stack space, you'll have to rewrite these words to use, perhaps, a temporary array.

The final step in creating a new cursor is shown in block 105. `CLOCK-MASKS` is the array which will hold the masks. On line four, the screen mask is loaded onto the stack from block 106 (you can see the outline of the clock in the one/zero pattern). The address of the array is then put on top, and the screen mask is stored. On line five, the same is done for the cursor mask of block 107. `CLOCK-CURSOR`, when executed, redirects the mouse drivers to use the new clock array, with the hot spot in the center (pixel 8,8).

Returning to the arrow cursor is done by initializing the mouse with `IMOUSE`. This, however, turns off and centers the cursor. To make the switch back to the arrow seamless, `ARROW-CURSOR` (block 104) first calls `MOUSE?`, stacking the current position. It then calls `IMOUSE`, restores the cursor position using `MOUSEAT` and the stacked coordinates, and redisplays the cursor.

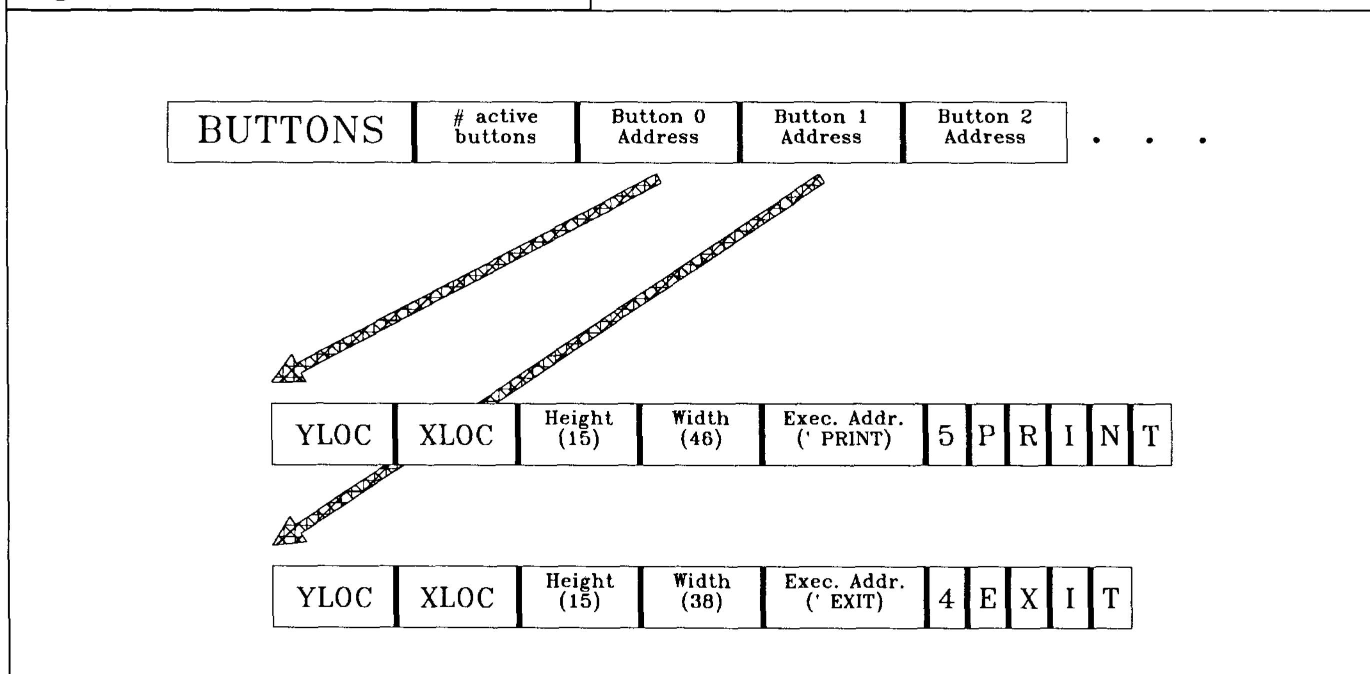
Button Words

The button words, like the mouse words, are simple and straightforward. However, a solid understanding of the philosophy and software architecture behind the code is probably necessary for a real grasp of the system.

When I set out to develop this system, my goal was to have a way of establishing (or instantiating, for you computer science buffs) a button by supplying only:

a. the button's location on the screen,
b. the code to be executed by the button, and
c. the name of the button.

Figure Two. Button-software architecture.



In addition, I wanted a single word to display all of the buttons, and another word to monitor button clicks and execute the code associated with a button.

The architecture used to accomplish this is illustrated in Figure Two. Each button is essentially an object. The data structure representing a button contains all of the information necessary to draw the button, check to see if it is clicked, and execute the code associated with the button. Figure Two contains data structures representing buttons called "Print" and "Exit." The first two cells of these structures contain the Y and X coordinates of the button's upper-left corner. The next two cells contain the height and width of the button. The data in these four cells is used both to draw the button, and to determine whether the cursor is positioned on the button. The fifth cell contains the address to be executed should the button be clicked. Following this is a counted string containing the name to be printed on the button.

A button's height and width are dependent on the name of the button. polyFORTH's EGA graphics generate characters that are 11 pixels tall by eight pixels wide. Two blank pixels above and below the name make a very presentable-looking button. Thus, the height of a button will always be 15 pixels. A three-pixel buffer in front of and behind the name also looks good. Thus, "Print" will be (5 * 8) + 6 (that's `5 8 * 6 +`) or 46 pixels wide. "Exit" will be 38 pixels wide. The minimum size of a button data structure is 12 bytes. The maximum size depends on the length of the name.

Notice that these data structures are headless. Having single words to both display the buttons and check for clicks requires that these words be able to cycle through the button data structures, doing the necessary work with the data from one button, and then moving on to the next. Thus, the addresses of all the structures must be stored and managed somewhere. This is done using a higher-level data structure named `BUTTONS`. The first cell of `BUTTONS` contains the number of buttons in the system. Each cell following this contains the address of one button. Thus, the words which display and check the buttons can be written with `DO` loops, working their way up this data structure, visiting each of the buttons. Since the button data structures are accessed in this indirect fashion, heads are unnecessary. In fact, they don't lend anything useful to this application.

Block 42 contains the words to set up, manage, and access the `BUTTONS` data structure. On line four, `BUTTONS` is created. Note that it is `MAX#-BUTTONS 2 * 2 +` long, providing for, in this case, nine button addresses plus the button count. `INIT-BUTTONS` initializes `BUTTONS` by storing zeroes in each cell. `ADD-BUTTON` accepts the address of a button data structure off the stack, and stores it in the next available cell of `BUTTONS`, incrementing the count. If `BUTTONS` is full, `ADD-BUTTON` says so and does nothing.

Block 43 contains the code to create a button and its data structure, and to access the data contained in a button's data structure. Note the usage of `BUTTON`. Let's say we wanted to make a button named "Hello" which printed out "Hi." First, we could write:

```
: TEST ( --) ." Hi." ;
```

Then we could create the button with:

```
100 200 ' TEST BUTTON Hello
```

The word `BUTTON` first puts the current dictionary pointer location on the stack with `HERE`. This is the address of our soon-to-be-created data structure. It adds this button to `BUTTONS` by giving a copy of the address to `ADD-BUTTON`. It then moves this button's address, along with the executable address, to the return stack and lays down the Y and X coordinates of the button location. Next, the height, 15, is laid down, followed by a zero as a space keeper for the width. Finally, the executable address is laid down, followed by the name string. Last, the string count is fetched back out from its location ten bytes above the button's base address. The button's width is then calculated and stored back over the place holder, zero. With that, a new button data structure exists in memory, with the latest cell of `BUTTONS` pointing to it.

The data access words on block 43 are used to pull the necessary information out of a button's data structure. `>BUTTON` takes a button number (0, 1, 2, ...) and returns the address of that button's data structure. Note that if you provide a button number that doesn't yet exist, you'll get a nice surprise. `BUTTON-LOC` accepts a button's address, and returns the Y and X coordinates of the button. `BUTTON-SIZE` returns its dimensions. `DO-BUTTON` executes the code associated with a button.

Block 44 contains all the words used to display a button. `.BUTTON-NAME` accepts a button's address and prints the button's name at the correct location on the screen. `.BODY` displays a rectangle with a white border at the button's location. `.BUTTON` displays a button by first drawing the button's body, a red rectangle with white border, and then printing the name on it in white. `.RBUTTON` draws the button in reverse, a white rectangle with red lettering. This is used later on to provide a visual cue that a button is pressed. `.BUTTONS` displays all of the active buttons. This is typically used only during the initialization of the GUI, when the graphics screen is being set up. Note how simple this word is.

Block 45 contains the words used to determine whether a given set of coordinates (from the mouse) lies on a button. `XRANGE` and `YRANGE` accept a button's address and return the pixel values corresponding to the button's left and right, or top and bottom edges. `ON-BUTTON?` accepts the three values returned by the `?PRESS` mouse word, along with a button address. If the correct mouse button is pressed ***and*** the cursor is on the button, this word returns true. `UP-WAIT` creates a pause until the mouse button is released, then exits. This provides a means for executing a clicked button's code after the mouse button is released, rather than when it is pressed.

Finally, block 46 is where it all happens. The word `DO-BUTTONS` is typically used in a `BEGIN...UNTIL` loop looking something like this:

```
: MY-GUI ( --)
  BEGIN DO-BUTTONS ?KEY UNTIL ;
```

DO-BUTTONS first calls ?RELEASE for the left mouse button, clearing this function. It then calls ?PRESS for the left mouse button. Using the information returned by ?PRESS, a DO loop is entered calling ON-BUTTON? for each active button in the system. If ON-BUTTON? for a given button returns false, nothing happens. Should ON-BUTTON? return true, this button is being clicked. Given that, the first order of business is to show the user the button is clicked. The cursor is turned off (this should always be done before drawing—drawing over the cursor yields strange results). The reversed button is then displayed and the cursor is turned back on. UP-WAIT is then entered. At this point, the user sees a reversed button on the screen and can move the mouse wherever he wants. Nothing will happen until he releases the mouse button. After the mouse button is released, the button is redrawn normally and DO-BUTTON executes the button's code. And with that, our voyage is complete.

I have used these words in a number of applications, and have found them to be very robust and complete. An example of one of these applications is the Fourier analysis package illustrated in Figure Three.

Block 47 contains a simple example of the system in use. Rather than using a keystroke to exit the GUI, a variable is tested. This way, the button named "Exit" can modify the variable from within DO-BUTTONS, and take us out of the loop—just the way the big boys do it. The GUI generated by the words on block 47 is illustrated in Figure Four.

Alterations/Alternatives

The software architecture illustrated in Figure Two is simple, straightforward, (Have I been using that phrase a lot? Good.) and gets the job done. However, the number of buttons is limited by the size of the BUTTONS data structure, and should the number be less than the maximum, memory space is wasted (it may only be a few bytes, but some systems can use those bytes). An expandable system could be made by using a linked list of button data structures, rather than maintaining an array of pointers to them. The architecture of this type of system would look something like Figure Five. Each button data structure would contain an additional two-byte field for the link. The first (bottom) button would contain a link address of zero. The link field of the next button contains a pointer to the link field of the bottom button. The third button's link field points to the second button's, and so on. The address of the topmost button's link field is kept in the variable LAST-BUTTON.

In the system with the BUTTONS data structure, words which must traverse the button list do so using a DO loop and a count. In the linked list system, button traversal is done by using an uncounted BEGIN...UNTIL loop, fetching the next link address by using the current one, and testing each address until a zero link is fetched, indicating that the bottom of the list has been reached. This obviously changes the software a bit. Block 42 is completely discarded, and the variable LAST-BUTTON is created. The word BUTTON can be easily altered to lay down the link to the topmost button, and store the address of the latest button in LAST-BUTTON. Also, the data-access words on block 43 must all have their offsets altered to work from the link field address. The button-traversing .BUTTONS and DO-BUTTONS words must, of course, be altered to use a BEGIN...UNTIL loop.

This architecture is more elegant and uses a block less

Figure Three. The mouse-and-button words in use.

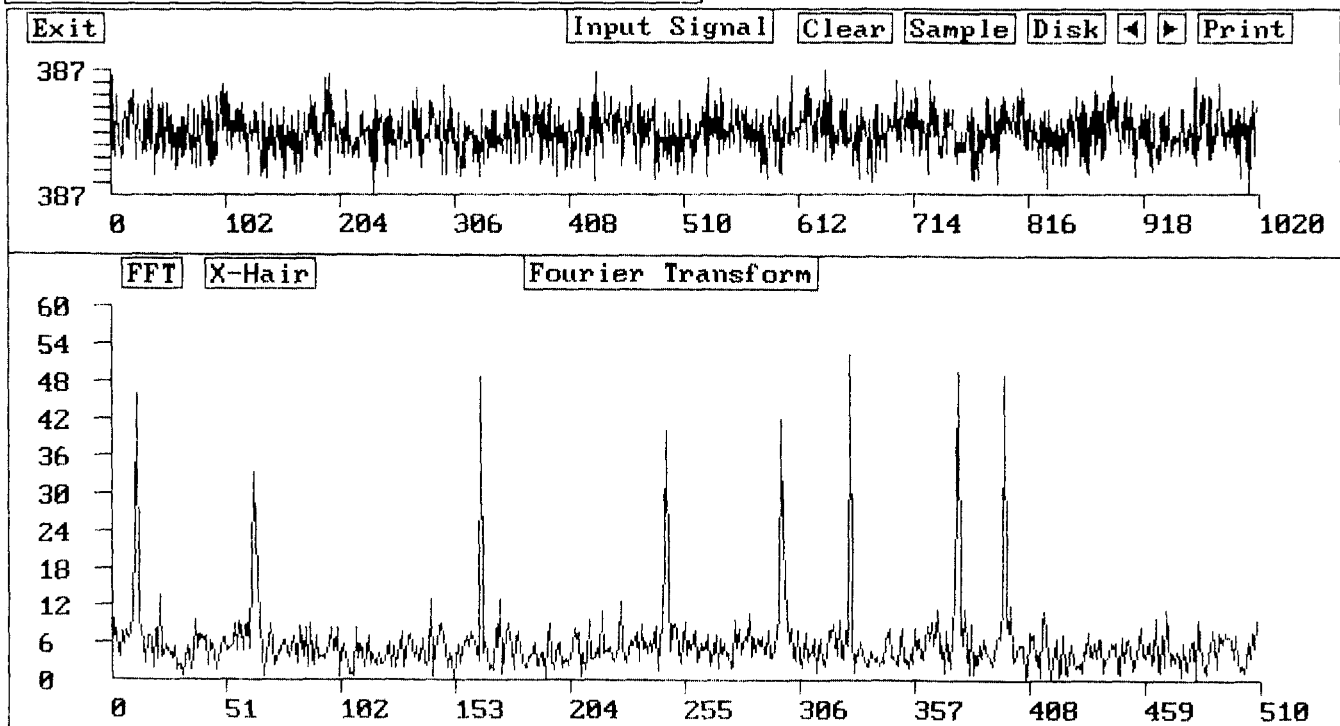


Figure Four. The GUI generated by block 47.

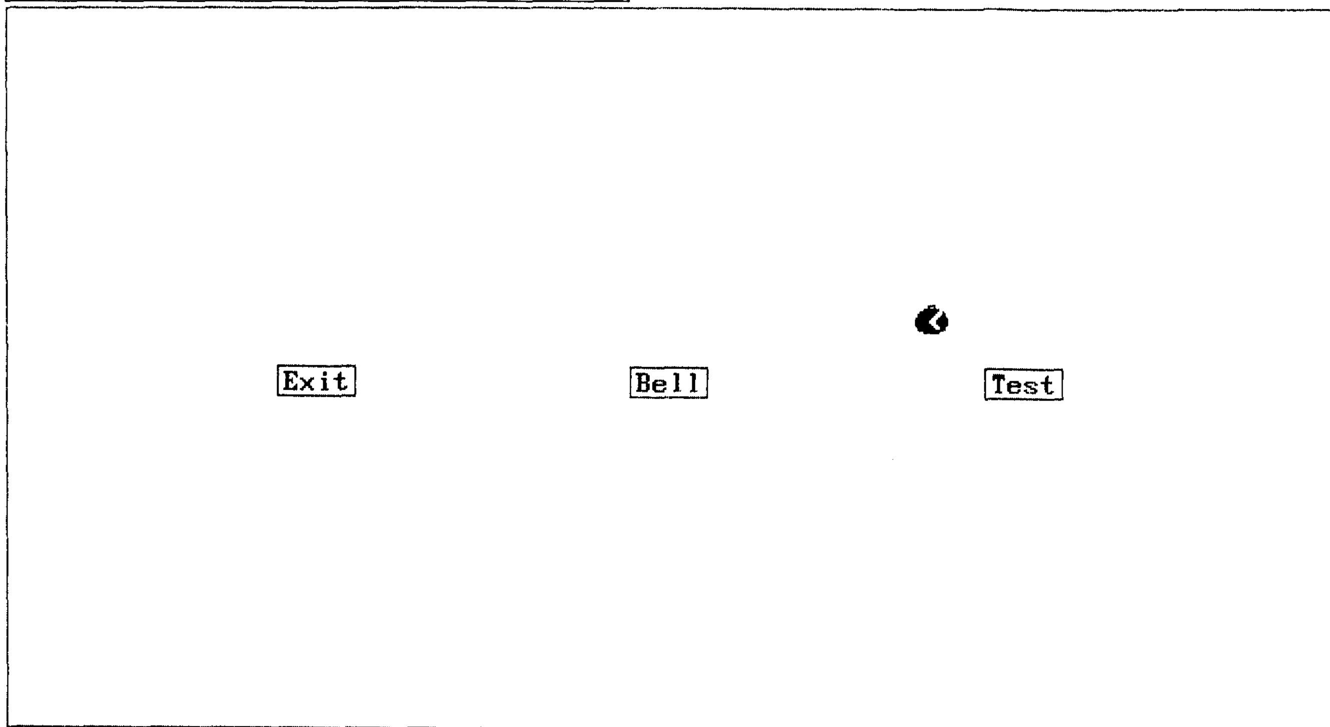
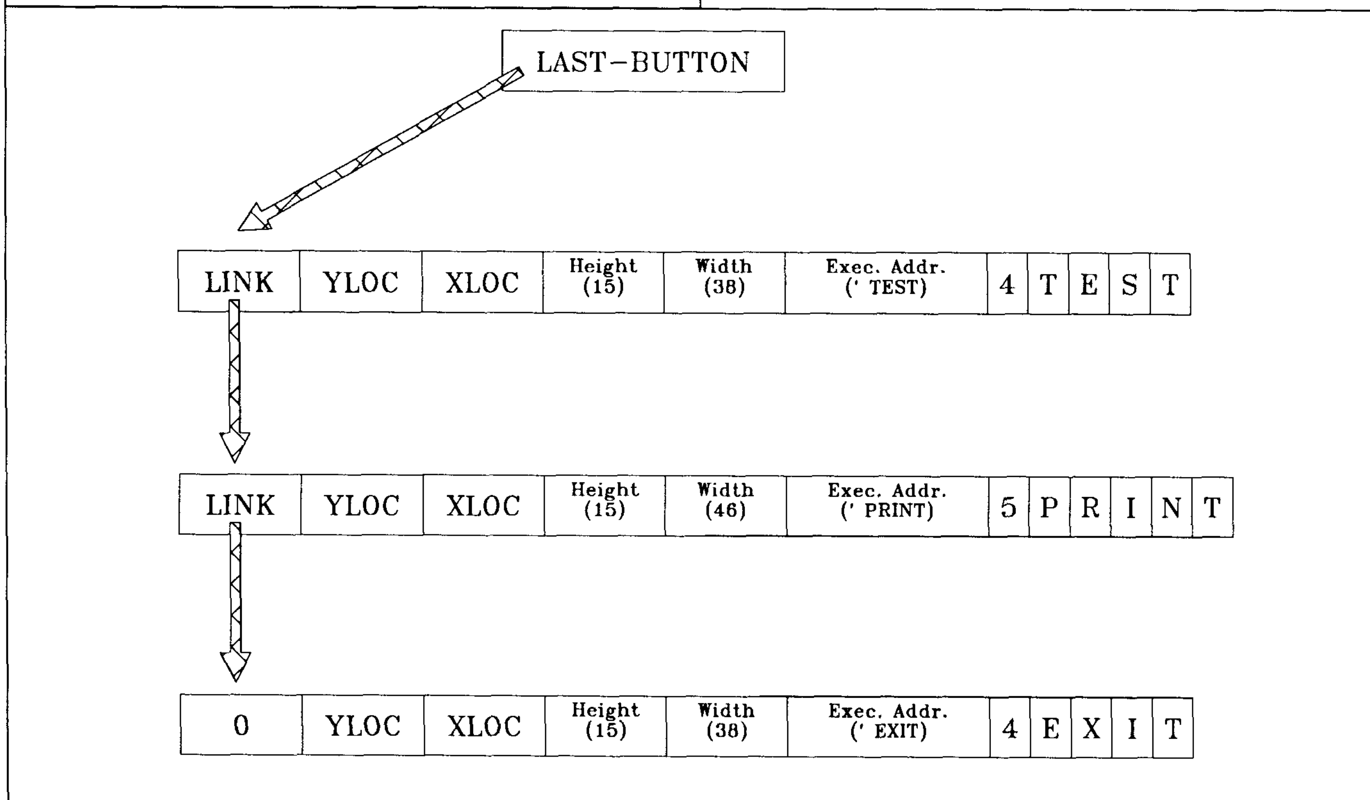


Figure Five. Linked-list button-software architecture.



code than the original. It will run at about the same speed as the original system, far faster than the operator can detect. In the end, personal preference will probably determine which system you will use.

The word BUTTON, as it was originally written, accepts a string for the name to be printed on a button. BUTTON could be rewritten to instead print the name of the executable word to which a particular button is connected. I wasn't interested in this kind of functionality. Instead, I wanted to be able to use any ASCII string for the name of the button. In Figure Three, you can see that I used a set of arrow buttons to move back and forth through the input signal.

The Benediction

Remember that anything someone else can do on a computer, you can do too. There is no magic in this universe, and since you're using Forth, you can probably do about ten times what those other programmers can do—if you address *The Real Problem.* I'll leave you with this:

Creativity is more than just being different...
Anybody can play weird—that's easy.
What's hard is to be as simple as Bach.
Making the simple complicated is commonplace...
Making the complicated simple—awesomely simple;

That's creativity.

—Charles Mingus

References

1. *DOS Programmer's Reference,* 3rd Ed.; Terry Dettmann, Jim Kyle, and Marcus Johnson; 1992. Que Corporation, 11711 N. College Ave., Carmel, IN 46032.

Rich Wagner is an aerospace engineer and computer programmer living in Revere, Pennsylvania. He began using Forth in 1989 while developing the Sensor Driven Airborne Replanner, an RTX2000- and CMForth-based robotic aircraft control system for the U.S. Navy's Unmanned Air Vehicles.


```
BLOCK 41
 0 ( GRAPHIC INTERFACE - COORDINATE TRANSFORMATIONS )
 1
 2 CODE XFORM ( Y,X--Y',X') 0 POP 1 POP 349 # W MOV
 3                   1 W SUB W PUSH 0 PUSH NEXT
 4
 5 : PEL ( Y,X--) XFORM PEL ;
 6 : RULE ( Y1,X1,Y2,X2--) 2>R XFORM 2R> XFORM RULE ;
 7 : PFAT ( Y,X--) AT ;
 8 : AT ( Y,X--) XFORM AT ;
 9
10
11
12
13
14
15

BLOCK 42
 0 ( GRAPHIC INTERFACE - BUTTONS )
 1
 2 9 CONSTANT MAX#-BUTTONS
 3
 4 CREATE BUTTONS    MAX#-BUTTONS 2* 2+ ALLOT
 5
 6 : INIT-BUTTONS ( --) BUTTONS MAX#-BUTTONS 2+ ERASE ;
 7
 8 : #BUTTONS? ( --N) BUTTONS @ ;
 9
10 : ADD-BUTTON ( a--) #BUTTONS? MAX#-BUTTONS =
11              IF ." Buttons full." DROP
12            ELSE #BUTTONS? DUP 2* 2+ BUTTONS + ROT SWAP !
13                 1+ BUTTONS !
14            THEN ;
15                                         INIT-BUTTONS

BLOCK 43
 0 ( GRAPHIC INTERFACE - BUTTON DATA STRUCTURE AND ACCESS )
 1
 2 : BUTTON ( YLOC,XLOC,a--) HERE DUP ADD-BUTTON >R >R SWAP
 3          , , 15 , 0 , R> R> SWAP , 32 STRING DUP 10 +
 4                        C@ 8 * 6 + SWAP 6 + ! ;
 5
```

```
 6 ( ----------USED: YLOC XLOC ' Name BUTTON String-----------)
 7
 8 : >BUTTON ( N--a) 2* 2 + BUTTONS + @ ;
 9
10 : BUTTON-LOC ( a--Y,X) DUP @ SWAP 2+ @ ;
11
12 : BUTTON-SIZE ( a--dY,dX) DUP 4 + @ SWAP 6 + @ ;
13
14 : DO-BUTTON ( a--) 8 + @EXECUTE ;
15
```

BLOCK 44

```
 0 ( GRAPHIC INTERFACE - BUTTON DISPLAY )
 1
 2 : .BUTTON-NAME ( a--) DUP BUTTON-LOC XFORM 1 3 V+ PFAT
 3        10 + DUP C@ 0 DO 1+ DUP C@ PLACE LOOP DROP ;
 4
 5 : .BODY ( a--) DUP BUTTON-LOC XFORM 2DUP PFAT ROT BUTTON-SIZE
 6               2DUP RECTANGLE 2SWAP PFAT WHITE BOX ;
 7
 8 : .BUTTON ( a--) DUP RED .BODY WHITE .BUTTON-NAME ;
 9
10 : .RBUTTON ( a--) DUP WHITE .BODY RED .BUTTON-NAME ;
11
12 : .BUTTONS ( --) #BUTTONS? 0 DO I >BUTTON
13                                    WHITE .BUTTON LOOP ;
14
15
```

BLOCK 45

```
 0 ( GRAPHIC INTERFACE - BUTTON POLLING AND CONTROL )
 1
 2 : XRANGE ( a--Xl,Xh) DUP BUTTON-LOC XFORM SWAP DROP
 3                SWAP BUTTON-SIZE SWAP DROP OVER + ;
 4
 5 : YRANGE ( a--Yl,Yh) DUP BUTTON-LOC XFORM DROP SWAP
 6                          BUTTON-SIZE DROP OVER + ;
 7
 8 : ON-BUTTON? ( X,Y,t,a--t) >R
 9        IF  R@ YRANGE WITHIN SWAP R> XRANGE WITHIN AND
10       ELSE 2DROP R> DROP 0 THEN ;
11
12 : UP-WAIT ( --) BEGIN 0 ?RELEASE >R 2DROP R> UNTIL ;
13
14
15
```

BLOCK 46

```
 0 ( GRAPHIC INTERFACE - BUTTON POLLING AND CONTROL )
 1
 2 CODE 3DUP ( N,N,N--N1:N3,N1:N3) S W MOV 4 W) PUSH 2 W) PUSH
 3                                   W ) PUSH NEXT
 4
 5 CODE 3DROP ( N,N,N--) 6 # S ADD NEXT
 6
 7 : DO-BUTTONS ( --) 0 ?RELEASE 3DROP 0 ?PRESS
 8    #BUTTONS? 0 DO
 9           3DUP I >BUTTON DUP >R ON-BUTTON? R> SWAP
10           IF   DUP -MOUSE .RBUTTON .MOUSE UP-WAIT
11                DUP -MOUSE .BUTTON  .MOUSE DO-BUTTON
12           ELSE DROP THEN
13    LOOP 3DROP ;
14
15
```

```
BLOCK 47
 0 ( BUTTONS -- GUI SETUP EXAMPLE )
 1
 2 VARIABLE ?EXIT
 3
 4 : GET-OUT ( --) -1 ?EXIT ! ;
 5 : DING-DONG ( --) BELL ;
 6 : TAKES-A-WHILE ( --) CLOCK-CURSOR 3000 MS ARROW-CURSOR ;
 7
 8 175 131 ' GET-OUT BUTTON Exit
 9 175 300 ' DING-DONG BUTTON Bell
10 175 470 ' TAKES-A-WHILE BUTTON Test
11
12 : GUI-INIT ( --) GR .BUTTONS .MOUSE 0 ?EXIT ! ;
13
14 : GUI ( --) GUI-INIT BEGIN DO-BUTTONS ?EXIT @ UNTIL -MOUSE ;
15

BLOCK 101
 0 ( MOUSE WORDS) HEX
 1 CODE IMOUSE ( --t,#B) 3 W MOV 0 # 0 MOV 33 INT
 2                       0 PUSH 3 PUSH W 3 MOV NEXT
 3
 4 ( RETURNS t IF MOUSE IS PRESENT, AND #BUTTONS )
 5
 6 CODE .MOUSE ( --) 01 # 0 MOV 33 INT NEXT
 7
 8 CODE -MOUSE ( --) 02 # 0 MOV 33 INT NEXT
 9
10 CODE MOUSE? ( --X,Y,B) 3 W MOV 03 # 0 MOV 33 INT
11                        1 PUSH 2 PUSH 3 PUSH W 3 MOV NEXT
12
13 CODE MOUSEAT ( X,Y--) 2 POP 1 POP 04 # 0 MOV 33 INT NEXT
14
15                                                    DECIMAL

BLOCK 102
 0 ( MOUSE WORDS) HEX
 1
 2 CODE ?PRESS ( 0/2/1--X,Y,#) 3 W MOV 3 POP 05 # 0 MOV 33 INT
 3             1 PUSH 2 PUSH 3 PUSH W 3 MOV NEXT
 4
 5 CODE ?RELEASE ( 0/2/1--X,Y,#) 3 W MOV 3 POP 06 # 0 MOV 33 INT
 6             1 PUSH 2 PUSH 3 PUSH W 3 MOV NEXT
 7
 8 ( ?PRESS & ?RELEASE RETURN # OF TIMES THE SPECIFIED BUTTON WAS )
 9 ( PRESSED OR RELEASED, AND THE LATEST LOCATION, SINCE LAST CALL)
10
11 CODE !XLIMS ( H,L--) 1 POP 2 POP 07 # 0 MOV 33 INT NEXT
12
13 CODE !YLIMS ( H,L--) 1 POP 2 POP 08 # 0 MOV 33 INT NEXT
14
15                                                    DECIMAL

BLOCK 103
 0 ( MOUSE WORDS - CURSOR DEFINITION )
 1
 2 VARIABLE MASK-DEST
 3
 4 : !ROW ( 16 0's OR 1's,ROW#,a--) >R >R 0
 5              16 0 DO 2* OR LOOP R> 2* R> + ! ;
 6
 7 : IMASKS ( a--) 32 0 DO DUP 0 I 2* ROT + ! LOOP ;
 8
 9 : !SCREEN-MASK ( 256 0's OR 1's,a--) MASK-DEST !
10                    0 15 DO I MASK-DEST @ !ROW -1 +LOOP ;
```

```
11
12 : !CURSOR-MASK ( 256 0's OR 1's,a--) MASK-DEST !
13                  16 31 DO I MASK-DEST @ !ROW -1 +LOOP ;
14
15
```

BLOCK 104

```
 0 ( MOUSE WORDS - CURSOR DEFINITION ) HEX
 1
 2 CODE SET-SHAPE ( HOTX,HOTY,a--) 2 POP 3 W MOV 1 POP 3 POP
 3                  09 # 0 MOV 33 INT W 3 MOV NEXT
 4
 5 DECIMAL
 6
 7
 8
 9 : ARROW-CURSOR ( --) MOUSE? DROP IMOUSE 2DROP MOUSEAT .MOUSE ;
10
```

BLOCK 105

```
 0 ( CLOCK MOUSE CURSOR)
 1
 2 CREATE CLOCK-MASKS  64 ALLOT
 3
 4 106 LOAD CLOCK-MASKS !SCREEN-MASK
 5 107 LOAD CLOCK-MASKS !CURSOR-MASK
 6
 7 : CLOCK-CURSOR ( --) 8 8 CLOCK-MASKS SET-SHAPE ;
 8
```

BLOCK 106

```
 0   1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1
 1   1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1
 2   1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1
 3   1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1
 4   1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
 5   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 6   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 7   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 8   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
11   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
12   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13   1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
14   1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1
15   1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1
```

BLOCK 107

```
 0   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1   0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0
 2   0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
 3   0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0
 4   0 0 1 0 0 1 1 1 1 1 1 1 1 1 0 0
 5   0 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0
 6   0 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0
 7   1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1
 8   1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
 9   1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1
10   1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1
11   0 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0
12   0 1 1 1 0 0 1 1 1 1 1 1 1 1 1 0
13   0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0
14   0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0
15   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

(Editorial, continued from page 4.)

On-line Forth connections—develop *and maintain* a comprehensive list of on-line Forth resources from around the world for both print and electronic distribution.

FIG Chapter vitality-meister—develop resources, tools, and skills that will enable local leaders to hold meetings no Forth user will want to miss, and which may even attract non-Forthers, with content rich enough to warrant reporting in *Forth Dimensions* and, better yet, in the technology sections of local newspapers.

FIG membership growth—work closely with the FIG office to support ongoing efforts to build its membership base, which will in turn enable us to expand our range of services and involvement in worthy projects.

Forth business news and gossip—tap into information about interesting Forth projects, new contracts awarded, product development, technical advances, hiring and promotions, companies adopting Forth, etc., and regularly feed the acquired data to *Forth Dimensions* to keep our readers informed.

Academia—collaborate with academic Forth users to develop a high-quality textbook which will provide the instruction universally needed by engineering students, for example, using Forth on class-related projects. Get the book published (perhaps through Fig Leaf Press), then promote its adoption by E.E. departments and sneak it into the occasional C.S. department.

Instruction—write a book that teaches practical Forth in terms of how to build applications; more than "this is a stack" but stopping short of metacompilation and pyrotechnics. Use ANS Forth and make sure the examples work with available implementations; encourage one or more vendors to provide inexpensive Forth systems that will work *verbatim* with the book's examples. Completing the book's exercises should make the reader a solid intermediate-level Forth programmer, if not yet seasoned in the field. (Prediction: this is not the easiest task on my list, but the author who pulls it off will have a major success, and will have done more to promote Forth than the original Forth issue of *BYTE* and *Starting Forth* combined.)

That is my wish list for today. Some of the items on it will require an intensive, one-shot effort, while others will need lower-level but long-term attention. Some can be accomplished by the rugged individualist, but others will succeed only as a result of teamwork and consensus (for which e-mail, and perhaps the occasional conference call, should suffice).

I do not underestimate the amount of dedication, enthusiasm, and hard work that it takes to accomplish any worthy endeavor. But it would not be overstating the case to say that such efforts are at the very heart of the international Forth community and of FIG, and that they will determine its future course. Please consider exercising your own leadership by accepting the challenge presented by a worthy cause.

—*Marlin Ouverson*
ouversonm@aol.com

...such efforts are at the very heart of the international Forth community and of FIG.

Forth Interest Group
Statement of Change in Financial Position
Apr 30, 1992 to Apr 30, 1993

ASSETS:	4/30/92	4/30/93	Change + = Increase - = Decrease
Current Assets:			
Money Market	33,956.22	23,740.48	-10,215.74
Checking	2,845.94	9,855.61	7,009.67
Pending Foreign Clearing	51.67	0.00	-51.67
Returned Checks Pending	110.00	0.00	-110.00
Total Current Assets:	36,963.83	33,596.09	-3,367.74
Inventory:			
Inventory at cost	24,600.57	16,280.00	-8,320.57
Total Inventory:	24,600.57	16,280.00	-8,320.57
Other Assets:			
Deposit, United Parcel Service	200.00	0.00	-200.00
Second Class Postal Account	174.51	161.10	-13.41
Accounts Receivable	1,285.50	500.00	-785.50
Equipment	0.00	5,826.02	5,826.02
Total Other Assets:	1,660.01	6,487.12	4,827.11
TOTAL ASSETS:	63,224.41	56,363.21	-6,861.20
LIABILITIES:			
Sales Tax	46.66	100.88	54.22
FD Dues Alloc to future months	30,289.20	29,526.10	-763.10
TOTAL LIABILITIES:	30,335.86	29,626.98	-708.88
Financial Reserve:	32,888.55	26,736.23	-6,152.32

(Letters, continued from page 6.)
rigidly defined (and therefore have fewer side-effects) than C.

The practice of incrementally coding and testing that is so simple in Forth attracts people who want to work this way (me!). With local data for each module held on the stack, modular testing of each word is valid even after the word has been incorporated into other, secondary words.

As an end user of software products, I am continuously verifying the correct operation of the system (including hardware). It should not matter what source language/paradigm was used. My comfort level improves with software generated in a high-level language but if the system works, that is all that is required.

Regarding Jim Mack's letter, point three in his suggestions, I agree that FIG and Forth programmers should stop discussing threading issues. Selecting token vs. subroutine vs. direct vs. indirect threading is a performance and memory usage design issue and not a feature of the language. It is important to know that a design decision can be made, but it is an implementation decision and not part of the Forth language. The performance improvement between indirect and subroutine threading can be the difference between a microprocessor using a two-sided or a four-sided printed circuit board.

Lastly, I take exception to one of Jim Mack's opening comments. If a company (even the main force behind Forth acceptance) chooses to target their resources at a niche market, then it is their risk, their choice, and their reward. I have spent the last few years trying to be all things to all people. Working at what you are good at and want to do is where the money and return are.

Yours truly,
Tom Saunders
Edmonton, Alberta, Canada

Random Erratum

Everett (Skip) Carter wrote to inform us of an error in his code that accompanied "Generation and Application of Random Numbers," FD *XVI/1,2:*

I was re-writing my R250 random number code in order to make it ANS compliant so that it can be part of the Forth Scientific Library and found an error.

In both R250 and R250D there is a line:

```
r250_index @ 248 > if 0 r250_index ! then
```

that should be:

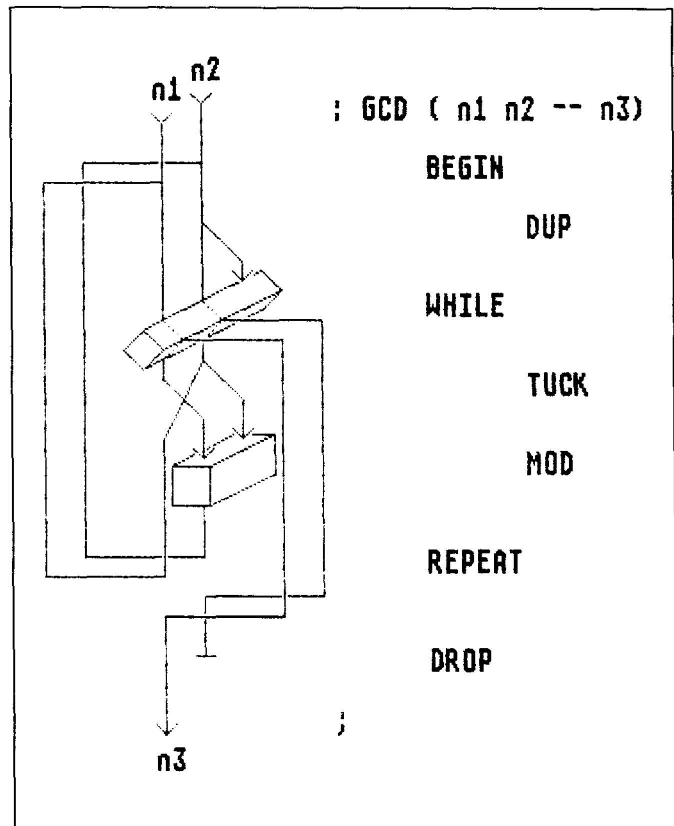
```
r250_index @ 249 > if 0 r250_index ! then
```

Pictures Worth a Thousand Comments

Dear Marlin,

One of my favourite Forth aphorisms is, "Forth is hardware by other means." With this is mind I devised, some time ago, a program development tool which allows me to visualise a colon definition as a kind of idealised circuit. I call it Stackflow.

Essentially, it is an extension of the redoubtable Wil Baden's control-flow diagrams which shows both control flow and data flow within one three-dimensional construct.



The example I give here is Euclid's diagram for greatest common divisor. I trust that it is quite self explanatory. The Stackflow metaphor has been of some benefit to me, as it gives a better handle on the stack than the classic "pile of plates in a cafeteria" metaphor. I hope it is of some interest to other FIGgers.

Yours faithfully,
Gordon Charlton
Hayes, Middlesex, United Kingdom

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

The Next Installment: Exposing Forth's Modules

In the last column I made considerable headway in showing that through its vocabulary mechanism, Forth has native namespace management facilities that can approximate a module or library mechanism—but nevertheless fall short of what is really needed.

In this installment I will elaborate upon other of the module-like ways of Forth. (Sort of like a game of "find the Pope in the pizza." Got it?)

A Compiler Module?

Leaving vocabularies aside, I think that Forth already contains at least one module, the collection of words that implements the Forth compiler.

There is a natural cohesiveness to the compiler words. For example, most of them have their immediate bit set and they all serve the purpose of helping compile routines.

True modules regulate the re-use of routines by establishing different levels of visibility for public and private routines. Forth asks the programmer to be self-regulating. "You should know better than to specify a compiler extension outside of the compilation of a routine."

With Forth as our foundation, we should use nothing less than state-of-the-art tools.

The compile-time behaviors of compiler extensions are limited to a single usage context, the compiling of a routine defintion. Whether a module system is proactive about making words invisible or the words themselves have a built-in limitation to reuse in certain contexts, the effect is similar.

The production of errors and undefined results underscores the need for more refined (dynamic) management of the visibility of Forth routines. Modules in general and a compiler module in particular can help manage the visibility of compiler routines so that their use is better regulated.

(Likewise, `POSTPONE` should eventually become a private routine of the `COMPILER` module.)

Not Diminished But Enhanced

If we use the visibility-management features of modules to prevent the misapplication of routines, the potential for successful routine reuse does not diminish. So modules don't have to produce the inefficient factoring that Leo Brodie feared (see chapter three of *Thinking Forth*).

Furthermore, we ought to use modules to tidy up the loose ends in an already well-optimized design, such as the design of Forth itself.

Perhaps in regard to its namespace management, Forth is still unrefined. However, there are considerable merits to Forth's unusual approach to it.

(Forth is both a compiler and an interpreter. It's not really all that unusual for interpreters to use some routine attribute in place of scope rules to distinguish search orders and routine visibility. For example, a `PATH` variable is often used in the command-line interpreters. Nevertheless, vocabularies can be considered somewhat unique.)

Because Forth vocabularies can be extended any time, they can be composed incrementally in separate compilation runs. Traditional modules are lexically delimited, so extending a module usually means recompiling the whole module from start to finish. (This is certainly true for C libraries.)

On the other hand, the mere existence of modules should help limit the need for annoying edit-compile-load cycles. A goal of modularization is the stabilization of one module at a time. So although lexically scoped modules are more trouble to manage, the idea is to design them only once, and then reuse them forever.

Them *and* Us Instead of Them *or* Us

Perhaps Forth can strike a compromise in order to obtain the benefits typically associated with lexically delimited modules, as well as the convenience of development of vocabulary-delimited modules. Inheritance and subclassing are a way to achieve such a blend of advantages in an object-oriented language. However, I don't think we have to buy into all the trappings of an object-oriented language to obtain the best of both mechanisms. More incremental refinements of vocabularies

should suffice.

The benefits of lexically delimited modules usually extend to simplified management of the code within a module. For example, it becomes easier to create operations that load and run a module, that release modules (freeing the associated memory), and that manage intermodule dependencies. An include-module operation is usually available to suppress repeated compilation of the same module yet ensure that module is compiled (or loaded) ahead of the modules that depend upon its services.

One of the disadvantages of vocabularies over modules was already claimed as an advantage. The ability of vocabulary-delimited modules to be defined in an interleaved and open-ended fashion tends to make the contents of the module less clear. This lets the programmer easily lose track of how the code has been modularized.

Certainly a fully featured modularization facility would instill incentives into Forth programmers to modularize their code more concretely. In previous columns, I urged the Forth community to adopt industry-standard linking and library technology. That's about as concrete as it can get. However, I am also amenable to Forth innovations.

As a first step toward a Forth-specific solution, I propose that module-supporting vocabularies be subdivided into privately and publicly visible namespaces with `INTERFACE`, a subvocabulary specifier.

Definitions placed in the `INTERFACE` subvocabulary of a normal vocabulary would be visible outside the parent vocabulary without making any reference to the parent vocabulary (providing that the module/vocabulary is already loaded or preloaded). Definitions not in the `INTERFACE` subvocabulary require the usual naming of the associated vocabulary to become visible. To add a definition to the interface portion of a vocabulary, however, both the vocabulary name and `INTERFACE` need to be specified, followed by `DEFINITIONS`.

Such a step falls far short of our need for substantive modularization tools. But it is only a first step. Many more coordinating refinements are needed. An important consideration is exactly how the interface portions of (loaded or preloaded) vocabulary modules are searched.

(Charles Moore has created a Forth that uses an `IMMEDIATE` vocabulary as a way to eliminate the `IMMEDIATE` attribute. We could likewise simplify Forth somewhat by treating as immediate all words that are placed in the `INTERFACE` subvocabulary of a `COMPILER` vocabulary.)

Regardless of their format, I hope you will learn to use full-blown modularization tools. With Forth as our foundation, we should not be using anything less than state-of-the-art tools. Let's bait Forthers and non-Forthers alike to go faster Forthward.

—Mike Elola
elolam@aol.com

Product Watch

APRIL 1994

Triangle Digital Services Ltd. announced the 40 megabyte and 20 megabyte TDS2020HD40 and TDS2020HD20 piggy-back boards for its TDS2020 SBC. These boards use the 1.3-inch Kittyhawk™ hard disk drive from Hewlett Packard. The drive features a matchbox form factor, glass media, automatic error detection and correction, and software-controlled spin-down modes. Power is further conserved by a large (up to a half-megabyte) static RAM buffer on the TDS2020 SBC. The size of the piggy-backed motherboard is 100x80x39 mm. (For two piggy-backed drives, the 39 mm. height increases.) A month of life can be obtained from a small battery with a system set up to conserve power.

Redirection of Forth output operations permits simple storage of data to a disk log. (Other data storage formats are supportable.) The retrieval of such data can be moderated through the serial port. Faster data transfer is possible through removal of the disk drive after field use. The TDS2020 SBC has additional connectivity options for graphics displays, a keyboard, and up to two PCMCIA/JEIDA boards, as well as serial communications links.

COMPANIES MENTIONED

Triangle Digital Services Ltd.
223 Lea Bridge Road
London E10 7NE
United Kingdom
Fax: 081-558 8110
Phone: 081-539 0285

Correction to ANSI Standard Forth Quick Reference

The stack diagram offered in FIG's quick reference for ANS Forth had at least one error. The stack diagram for the run-time operation associated with `S"` is incorrect, as is the description offered. (There is no compile-time stack diagram shown.)

The correct stack diagram is:

```
-- strAddr u
```

A more faithful description (or interpretation) is: Compile a string delimited by a double-quote character from the input stream and compile an operation that pushes the address of the first character of that string (n1) and its length (n2).

Please note these changes on your quick reference card so that you will not be confused by the error at some future time. Also, please send me (elolam@aol.com) notices of any other problems you encounter with the quick reference card so that subsequent editions of the card can be less misleading.

CALL FOR PAPERS

for the sixteenth annual and the 1994

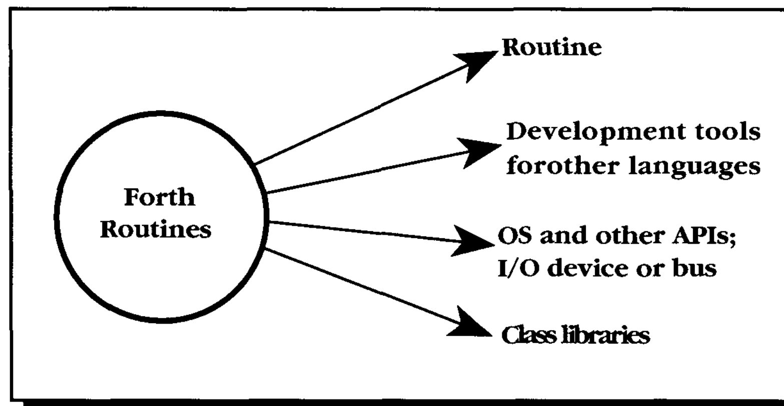
FORML CONFERENCE

The original technical conference for professional Forth programmers, managers, vendors, and users
Following Thanksgiving, November 25 - November 27, 1994
Asilomar Conference Center, Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California U.S.A.
Conference Theme:

"Interface Building"

Papers are sought that explore how code and data resources in various forms can be interfaced to maximize code reuse and programming efficiency.

Compiled routines represent the most fundamental code resources. The interface that makes it possible for compiled routines to work together so well involves a run-time system's call (return) stack and its parameter-passing mechanism. Nevertheless, exploiting their cooperative potential requires skillful programming. Each routine must be outfitted with just the right amount of functional scope (factoring), and with the correct choices of input and return parameters. How can this *interfacing art* be learned and fostered?



Libraries and modules have not been exploited well. In mainstream languages they offer only token support for managing related routines as (indivisible) collections that belong together. What are some possible treatments of Forth code that can establish more formal interfaces at the library-routine level or the module level?

Can interfaces be fashioned between Forth routines and the libraries, run-time systems, or data structures of other languages?

New programming languages keep appearing to tame various interfacing problems. Examples include Postscript, which establishes an interface around diverse printing engines so they can be treated similarly. Open Firmware (formerly Open Boot) wraps a standard environment around computer sub-system components, facilitating their configuration and initialization. X-Script and Telescript encapsulate multimedia and communications services, respectively. Among other things, they make it possible to view the same mail or multimedia item on disparate viewing platforms and over disparate, intervening networks. What common features do these interface-serving languages possess? Can an interface be constructed between Forth routines and the APIs and system call interfaces that serve as the compiled-language counterparts to these interface-serving languages?

Can Forth modules be crafted to let it talk to one or more I/O bus interfaces, such as those for PCMCIA, PCI, and "Plug N Play"?

How can Forth be interfaced to Windows, or equivalent GUIs? Besides linker technology, what is the most substantial obstacle that prevents our use of GUI-encapsulating class libraries such as MFC or OWL? Because SOM (system object model) attempts language independence, can it lead to a Forth interface to class libraries? What run-time interface provisions besides a call stack and a parameter-passing mechanism are going to be needed to support object-oriented Forths? To support event-driven programming?

Completed papers are due November 1, 1994.

Registration fee for conference attendees includes registration, coffee breaks, notebook of papers submitted, and for everyone rooms Friday and Saturday, all meals including lunch Friday through lunch Sunday, wine and cheese parties Friday and Saturday nights, and use of Asilomar facilities.

Conference attendee in double room - **$400** • Non-conference guest in same room - **$280** • Children under 18 years old in same room - **$180** • Infants under 2 years old in same room - **free** • Conference attendee in single room - **$525**

••• *Forth Interest Group members and their guests are eligible for a ten percent discount on registration fees.* •••

Mike Elola, Conference Chairman — Robert Reiling, Conference Director

Register by calling, fax or writing to:
Forth Interest Group, P.O. Box 2154, Oakland, CA 94621, (510) 893-6784, fax (510) 535-1295
This conference is sponsored by FORML, an activity of the Forth Interest Group, Inc. (FIG).