GETB and PUTB

Math—Who Needs It?

Charles Moore's Fireside Chat

One-Screen Unified Control Structure

Optimizing in BSR/JSR-Threaded Forth
FAST Forth Native-Language Embedded Computers

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.
- 6-channel I/O bus & 3 timer/counters.

SC/FOX SBC32 (Single Board Computer)
- 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 PCS (Parallel Coprocessor System)
- RTX 2000 industrial PGA CPU; 8 & 10 MHz.
- System speed options: 8 or 10 MHz.
- 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 SBC (Single Board Computer)
- RTX 2000 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.

For additional product information and OEM pricing, please contact us at:
SILICON COMPOSERS INC 208 California Avenue, Palo Alto, CA 94306 (415) 322-8763
## Features

<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>GETB and PUTB</td>
<td>Hank Wilkinson</td>
</tr>
<tr>
<td>9</td>
<td>One-Screen Unified Control Structure</td>
<td>Gordon Charlton</td>
</tr>
<tr>
<td>14</td>
<td>Charles Moore's Fireside Chat</td>
<td>C.H. Ting</td>
</tr>
<tr>
<td>18</td>
<td>Numbers</td>
<td>C.H. Ting</td>
</tr>
<tr>
<td>21</td>
<td>Optimizing in BSR/JSR-Threaded Forth</td>
<td>Charles Curley</td>
</tr>
<tr>
<td>27</td>
<td>Math—Who Needs It?</td>
<td>Tim Hendtlass</td>
</tr>
</tbody>
</table>

## Departments

<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Editorial</td>
<td>Forth consortium, numeracy, windows, &amp; on the stack.</td>
</tr>
<tr>
<td>4</td>
<td>Advertisers Index</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Letters</td>
<td>Strength mistaken; Volvos drove him to Forth, &amp; Forth's missing link; Embedded systems conference; Kelly's comparisons clarified.</td>
</tr>
<tr>
<td>39</td>
<td>Fast Forthward</td>
<td>From on-line discussion to hard-copy correspondence, the Forth community is developing a collective voice. Here's a digest of what it has said lately.</td>
</tr>
<tr>
<td>41</td>
<td>Volume XIII Index</td>
<td>A subject index to <em>FD</em> volume XIII. Back issues still available!</td>
</tr>
<tr>
<td>42–43</td>
<td>On the Back Burner</td>
<td>While you are getting the parts and assembling the board described in the last issue, our columnist takes time to explain metacompilation terminology and to foster on-line interaction.</td>
</tr>
</tbody>
</table>
Editorial

What if Forth businesses and associations teamed up to improve general awareness of Forth? They'd have to do something they all could agree on, benefit from, and contribute funds for. Something like a public-service ad for trade journals: "Sure, Forth fosters innovation by enabling programmers to explore highly personalized methods of problem solving. Some of your best people probably use it already, or know something about it—no coincidence. But did you also know that today's Forth systems can accommodate the rigorous methods and conventions of well-managed programming teams? That multi-tasking and metacompile are no problem—never have been—and that Forth can stand alone on its own considerable merits or peacefully co-exist with an operating system? Forth is a frequent flyer on the space shuttle, but also excels in earthbound applications like observatories, industrial automation, embedded controllers, medical/scientific instrumentation, and benchtop environments, not to mention consumer applications. Write or call for a free brochure and list of participating businesses..." Or, if funds were scarce, one might only be able to promote one of those low-brow, stick-in-your-mind jingles: "Go Forth, and your computer will say OK!"

Do you suffer from innumeracy in Forth, or just need a touch-up to your understanding about how to deal with digits? Dr. Ting's "Numbers" tutorial encourages beginners with the power of integer arithmetic. But if you need more than a beginner's dose, Prof. Tim Hendtlass' "Math—Who Needs It?" will further your understanding of different math packages, and will help you to choose the right routines—kindly provided—for the right jobs. (Hint: it's another instance in which too much power can corrupt performance.)

Speaking of performance, Windows makes an appearance in this issue. Forth for Windows has been implemented by two developers that we know of, Laboratory Microsystems, Inc. and Harvard Softworks. But for the determined, do-it-yourself hacker or the doggedly curious, not a lot has been forthcoming. Well, there's nothing like starting at the beginning, which would have to be reading and writing characters in the Windows format; see Hank Wilkinson's "GETB and PUTB" to get started.

One time, a hacker thought Forth had suffered long enough as a skeletal system with little help and no protection for the naive user. Thus was born a newer and better Forth with, among other things, a fully fleshed-out, interactive help and error-handling subsystem that relied on a separate stack to manage the many system-message strings. It was automatically invoked by the lower-level word HEY! (as in, "Hey, you clutz!?" every time a user did something unexpected. But the system died in beta testing when a couple of Forth gurus agreed, "Serious programmers will find it hard, being needled by a HEY! stack." [SFX: rim shot, groans]

...just a reminder... We greatly value the continued participation of each reader and FIG member, so please renew by mail, telephone, or fax at your earliest convenience. At the same time, consider giving a subscription to Forth Dimensions to a business, library, or colleague. We will look forward to sharing with them—and with you—the good work of the Forth community.

—Martin Ouverson

On the stack...

A line editor & history function
Forth-user profile
Application success stories
Forth in search of work
Integer date calculations
Build an 8051 metacomplier
ANS Forth: progress, analysis, and impact
Forth interface for the GP1B
(general-purpose interface bus)
...and much more!

Advertisers Index

Asian Business Contents .................. 40
Computer Journal ........................ 34
Forth Institute ............................ 44
Forth Interest Group ...................... 6, centerfold
Harvard Softworks ......................... 17
Laboratory Microsystems .................. 24
Miller Microcomputer Services ............ 11
Ofcite Enterprises .......................... 20
Silicon Composers .......................... 2

Forth Dimensions
Volume XV, Number 6
March 1993 April

Published by the
Forth Interest Group

Editor
Martin 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 © 1993 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 enjoys 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 $0/95/52 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to Forth Dimensions, P.O. Box 2154, Oakland, CA 94621."
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!

Strength Mistaken

Dear Editor,

"A Lesson in Economics" by Russell L. Harris (On the Back Burner, FD XIV/5) was a welcome piece of additional ammunition I'll be able to use the next time I'm nagged to use a DOS machine for an embedded or instrument-control application. Mr. Harris hits squarely on several key problems that crouch waiting to pounce on the naive IBM PC enthusiast who falls for the mirage.

On another note, along with Walter J. Rottenkolber, whose letter to the editor appeared in the same issue, I too have to disagree with Mike Elola on his article (FD XIV/4) about styling Forth to preserve the "expressiveness" of C.

Part of Forth's strength will always lie in its simplicity. However, this same simplicity is wrongly viewed as a weakness when it lets a programmer write unreadable code. The lack of expressiveness Elola refers to (that is, the lack of clarity as to where the stack cells come from, what they are, and what consumes them) is due to how the words are arranged in the source code just as much as to poor commenting; and yet, so often I see code written such that the breaks between lines, the space between words, and the words' starting columns have almost nothing to do with what the code is supposed to do. I even see things like a BEGIN in the middle of a line, with its corresponding UNTIL buried somewhere in the middle of another line and starting in a different column. Mr. Elola's indenting may help, but I fear this will constitute overuse of indenting, defeating much of the purpose of indenting, which is to make structures and program flow more obvious.

I'm tempted to write an article about writing readable code. I believe Mr. Elola has correctly identified a common problem. What I don't agree with is that it's a weakness of Forth itself. There are several things about C that I hope I have left in my past for the most part, and those include piles of parentheses and punctuation.

A third subject matter—What's happening with those ultra-fast stack micromprocessors I hear a little bit about here and there? One blurb I read recently cited 100+ MIPS (and Forth MIPS, at that) at a relatively low cost. This is certainly something I would expect to see get a lot of attention in EDN, Computer Design, and other trade magazines; yet I haven't seen a thing in those.

I was glad to see FIG would be at the Embedded Systems Conference (see letter below).

Sincerely,
Garth Wilson
11123 Dicky Street
Whittier, California 90606

Thanks for your comments, Garth. Please do write that article about readable Forth code—it continues to be important. The most thorough treatment I recall was by Kim Harris, whose paper in a Forth journal was presented several years ago at a FOML conference, and FIG distributes a cumulative index to FD articles, which contains some references to Forth style (see the first two issues on the mail-order form). But, as evidenced by much Forth code, those ideas either were not distributed well or were not adopted widely for some reason. Your further treatment of the subject might help.

As to stack-oriented CPUs, we welcome press releases about real products, articles by developers, and the experiences of users—as would, I presume, other publications like those you mention. Meanwhile, check out the "More on Forth Engines" series on the FIG mail-order form in this magazine. —Ed.

Volvos Drove Him to Forth,
& Forth's Missing Link

I have just returned from the Silicon Valley FIG Chapter meeting—have not missed more than four or five of them since June 23, 1990, when I first signed up. I enjoyed a chat with John and Frank Hall during the lunch break, and I want to follow up with the note I said I wanted to write to Forth Dimensions.

I am a mechanical engineer, have been designing cranes and heavy machinery for 30 years. I do not sing and dance like Leo Brodie, but I do drive old Volvos, and that is how I came to know Forth. You see, my wife and I started in computers when we bought our first Apple IIc; we started to look into the computer section at the library more often, and that is where, one day in April of 1990, the face of young Leo appeared on page v of the first edition of Starting Forth. I believe I had heard about Forth before that, probably through a Harris ad in one of the engineering trade magazines, and, seeing that a fellow who liked driving classic Volvos had written a book about it, I figured that Forth might not be all too bad.

Well, there is a long story of struggle, frustration, feelings of futility and defeat, but, even though I have not produced any masterpieces of Forth programming, I go to the meetings, I enjoy your magazine, I preach Forth. I have even gone to the torture of taking a C class at the local college, to see how bad things can be on the other side.

There are two reasons for me wanting to write this note:

First, I wish to thank you for publishing Olaf Meding's fine article, "Forth-Based Message Service." I understood it, liked it, and would like to see more articles with the same Fogg Index (or should I say "Fig Index")? I have a problem with the arcane and esoteric articles that remind me of my early struggles with De Bello Gallico, organic chemistry, Laplace transforms, etc.

Secondly, I have to voice my concern that there is
something missing in the Forth community that would attract newcomers, or I would not still be the novice of the group almost three years after I joined. It seems that we can cater only to the seasoned Forth programmers, perhaps to some degree to other programmers, but beginners cannot find any "Forth kits," as I would like to call them, that are inexpensive, readily available, work, have good documentation, and allow one to experiment and create without frustration. I think there is a need for a Forth interpreter package that can compete at least with the likes of GWBasic, in terms of size, availability, documentation, graphics, and floating-point math. Have I missed it somewhere along the way?

Sincerely,
Henry Vinerts
36139 Chelsea Drive
Newark, CA 94560

Erratum
Olaf Meding, author of last issue's "Forth-based Message Service," is employed by Amielco, 4800 Curtin Drive, McFarland, Wisconsin 53558; telephone 608-838-4194. The article was originally titled "To Boldly Go Forth Where No One Has Gone Before."

Embedded Systems Conference
Dear Mr. Ouverson,
I had the opportunity to browse the displays at the recent Embedded Systems Conference and see firsthand the hardware and software available. I was surprised at the number of sizzling, color-windowed, integrated C/C++ programming systems. Even the purveyors of Ada, the number two language there, tended to be a bit defensive. The once-ubiquitous BASIC was represented by only a couple of vendors. And Forth had only the Forth Interest Group waving the banner—a lonely island in the C's. Is the real world trying to tell us something?

These C compilers integrate an editor, syntax checker, compiler, and debugger that can work with either C or assembly source. Watch Expressions let you run C functions or display C variables interactively. In other words, these C systems seem to have a programming environment once exclusive to Forth. It would be interesting if someone familiar with both the new C compilers and Forth would compare them, especially regarding programming ease and productivity.

Programming controllers with the power systems would set you back $12,000 for software and hardware, and to this you would have to add a hefty computer. Most of the vendors' demonstrations used Sun workstations. But at the other end of the scale, ZWorld offers a line of 7980 controllers designed to be programmed with their $195 C compiler, which runs on a PC.

(Continued on page 16.)
GETB and PUTB

Hank Wilkinson
Greensboro, North Carolina

Working on my teaching certificate for high school physics at the University of North Carolina at Greensboro, it hit me that I want to use Forth in Windows when I teach. Since I learned my B. S. Physics previously at Guilford College (also Greensboro), only a year's study remains for my certification. This article describes simple commands found useful exploring Windows' files.

What do I mean by "use Forth in Windows"? Here is what I think I mean:

a) load Forth code contained in *.WRI files
b) write from Forth into *.WRI files
c) draw from Forth into *.BMP files
d) be "in" Windows when in Forth

Note that vectoring KEY and EMIT will not achieve any of the above goals. The first three goals require knowledge of the Windows data files. Simply put, the last goal requires knowledge of how Windows works. Frankly, a vendor could easily solve my dilemma.

Meanwhile, concepts described in Software Tools (Kernighan and Plauger; Addison-Wesley, 1976) help. Analogous to K&P's getc and putc, I have made GETB and PUTB. GETB reads exactly one byte from a file and leaves it on the stack. PUTB writes one byte from the stack into a file.

So you may follow, here is my system. My computer is a VSI PC '286 name-brand "compatible," with VGA, 40-meg hard drive, both a 5.25" and a 3.5" floppy, a mouse, HP Deskjet 500, a modem, and four megs of memory. I have DOS 5, Windows 3.1, and HS/Forth (regular—i.e., uses segmented memory).

I learned of HS/Forth and VARs in this journal, so will only review them. A VAR is a data structure with the behavior of both VARIABLES and CONSTANTS, while faster than either. CONSTANT-like, a VAR's value goes to the stack upon use. VARIABLE-like, the VAR's new value comes from the stack by placing 1S before the use of the VAR. Later uses of the VAR return its latest value.

Refer to the code at the end of this article. First we define TRUE and FALSE. Next, the handle and end-of-file-flag containers appear. Initializing the file's handle to TRUE (as opposed to FALSE) will not confuse an unopened file with the handle for the keyboard. For the end-of-file flag, logic dictates an unopened file has reached its end.

HS/Forth's MS-DOS system interface may be set to ABORT with a message upon error condition (FATAL), or pass the error on (INFORM). The words defined here assume HS/Forth will ABORT, giving immediate feedback.

To open a file for reading, we pass the address of the file's path\name to the HS/Forth word OPEN-R wrapped inside OPEN-GETB. Any error leaves G-H and G-EOF set FALSE. If successfully opened, G-H receives the file's handle and G-EOF receives TRUE.

To close the file, HS/Forth's CLOSEH is used. The file's handle goes on the stack for CLOSEH, which ABORTS upon any error condition. In that case, the VARs do not get touched. Otherwise, CLOSE-GETB closes the file and sets G-H and G-EOF to TRUE.

GET-BUF serves as the buffer for MS-DOS to put the byte read. HS/Forth's READH needs the memory segment, offset, number of bytes to read, and the handle. GETB encapsulates these functions and sets G-EOF. Notice that a successful read will return one byte. Reading past the end of file returns a zero (FALSE), and sets G-EOF TRUE.

Using GETB on, say, "c:\path\filename.ext," we first open the file. From the command line:

```
$ c:\path\filename.ext" OPEN-GETB[Enter]
```

does that. If successful,

```
GETB [Enter]
```

would display the byte's value, while

```
GETB EMIT [Enter]
```

would display the byte as an ASCII character. To determine if a byte read is valid, consider this code.

```
GETB G-EOF [Enter]
```

A zero (FALSE) displayed indicates a valid byte, while -1 (TRUE) shows the byte did not actually come from the file. Once finished playing, we issue:

```
CLOSE-GETB [Enter]
```

Now find the example code after the definition of GETB. A double number DVAR COUNTER holds a count of the number of bytes in a file. The routine COUNT-BYTES expects an opened file and proceeds counting bytes until G-EOF becomes TRUE.

Notice the test for the end of file inside the loop yields zero for an empty file or an unopened file. At any rate, COUNT-BYTES counts the bytes, while COUNT-FIILE performs administration.

Writing to a file is slightly simpler than reading. We only need a VAR to hold the F-H (put handle). OPEN-PUTB makes use of HS/Forth's MKFILE, which creates or erases an existing file. The handle passed by MKFILE goes into F-H.

CLOSE-PUTB is analogous to CLOSE-GETB.

PUTB stores the byte on the stack in the PUT-BUF and passes the memory segment, offset, byte count, and handle to WRITEH, which returns the number of bytes written.
Testing the actual number of bytes written serves as error checking.

For an example using PUTB, we show copying a file. Buffers for filenames make the process easier. The names GBS and PBS allow quick typing. In COPY-FILE, the user is shown the path/filenames from both buffers. Either a “Y” or “y” are required for copying to take place.

With a proper response, the corresponding files are opened and the copying—byte by byte—begins. As soon as the end-of-file flag is found TRUE, the copying stops and the files are closed. During copying, any key hit stops the process.

These routines are simple to use and understand, and are robust enough for use from the command line. Their simplicity allows easy modification. For something to get into use quickly, use them as is.

From time to time, the need for faster file-handling code becomes apparent. In that case, use larger buffers and design buffer handling. (I have spent more time optimizing code than I have saved by executing optimizing code. Routines shown here reflect that experience.)

GETB and PUTB form crucial elements of routines I used exploring Windows’ files. One routine counts the frequency of bytes. Another finds the occurrence of arbitrary byte patterns. A third routine creates a file dump. The ability to read and write one byte at a time is fundamental.

Hank Wilkinson and his wife of nineteen years have two children in high school and one in elementary school. He has been employed in the construction trade (five years) and in the wholesale supply business (thirteen years), and operated his own programming firm (five years). Currently, he is a graduate student working on public school teacher certification. He has used Forth for twelve years.

0 VAR FALSE
-1 VAR TRUE

TRUE VAR G-H \ Get-Handle storage
TRUE VAR G-EOF \ Get End Of File flag
\ TRUE = EOF, FALSE = not EOF

\ use: "$/path/filename* OPEN-GETB
OPEN-GETB ( address -- )
G-FILE ( addr -- handle )
IS G-H
FALSE IS G-EOF ;

\ use: CLOSE-GETB
CLOSE-GETB ( -- )
G-H CLOSEH
TRUE IS G-H
TRUE IS G-EOF ;

CREATE GET-BUF 1 ALLOT \ 1 byte buffer

\ use: GETB
GETB ( -- byte )
\ G-EOF TRUE, invalid file byte
\ G-EOF FALSE, valid file byte
LISTS $ GET-BUF 1 G-H REACH
0= IF TRUE IS G-EOF FALSE EXIT THEN \ file empty
GET-BUF C ;

0 S->D DVAR COUNTER

\ use: COUNT-BYTES
( file must be open, counter cleared

\ use: "$/path/filename* COUNT-FILE D.
COUNT-FILE ( a -- d )
\ n = offset of counted string to file’s \path\filename
CR ." Counting bytes in "DNP COUNT TYPE CR
."
Hit any key to stop." CR
0 0 IS COUNTER \ clear count to start
OPEN-GETB
COUNT-BYTES
CLOSE-GETB
COUNTER ;

TRUE VAR P-H \ Get-Handle storage

\ use: "$/path/filename* OPEN-PUTB
OPEN-PUTB ( address -- )
MFILE ( addr -- handle )
IS P-H ;

\ use: CLOSE-PUTB
CLOSE-PUTB ( -- )
P-H CLOSEH
TRUE IS P-H ;

CREATE PUT-BUF 1 ALLOT \ 1 byte buffer

\ use: 0 PUTB ( writes 0 to file )
PUTB ( byte -- )
PUT-BUF C !
LISTS $ PUT-BUF 1 P-H WRITEH
0= IF ." Write error! " EXIT THEN ;

\ FILE NAME HOLDERS
CREATE GBS 128 ALLOT \ GETB file name
CREATE PBS 128 ALLOT \ PUTB file name

\ initialize filenames to something
$" GETPUT.FTHU GBS $!
$" XX.FTHU PBS $!

\ use: ( filenames string variable already set )
COPY-FILE
COPY-FILE ( -- )
CR
." Copying file named "GBS $.
CR ." into file named "PBS $.
CR ." is this correct? (Y/y)"
KEY ASCII Y OVER = SWAP ASCII Y = OR
IF ." Okay, we're copying.
ELSE ." Not copying" CR EXIT THEN
CR ." Hit any key to abort COPY-FILE." CR

GBS OPEN-GETB
PBS OPEN-PUTB
BEGIN
GETB
G-EOF FALSE = WHILE
PUTB
?TERMINAL TP ." Quitting, so delete partial file." CR
CLOSE-PUTB CLOSE-GETB EXIT THEN
REPEAT
DROP \ drop spurious byte read when file was empty
CLOSE-PUTB
CLOSE-GETB ;
One-Screen Unified Control Structure

Gordon Charlton
Hayes, Middlesex, U.K.

This article was prompted by Kourtis Giorgio's Curly Control Structure Set. Giorgio stated that his intention was to include every good idea he had come across. This turned out to be a good many good ideas, so Giorgio has provided the archetypal Fat Forther's solution. Although I am not a devout minimalist, I do concur with the principal that less is more. As the primary design criterion here is brevity, I present the one-screen version in all its muck and glory.

Three Non-Solutions

The ultimate reductionist solution is ?GOTO. This is not a solution, as it is unstructured. It is demonstrable that a zero-tripping FOR NEXT can be coerced into sufficing, at the cost of outrageous ineffectiveness and complexity. This is not a viable solution. One can also get by with IF, THEN, and RECURSE. Although popular with the AI community, this solution does have certain problems with efficiency and readability. Therefore, this is not a solution either.

The Solution

Although three words do not cut the mustard, we will see that four words are enough. In fact, I will introduce two extra ones, for convenience and to remove a slight inefficiency. The Unified Control Structure (UCS) is derived from two previous proposals which do not appear in Giorgio's comprehensive bibliography: they are the Hainsworth Extended Case and the Universal Delimiter.

One Screen

When commencing a project, I will often attempt to come up with a solution that fits within one screen. This is a very rigid discipline and focuses the mind excellently. A lot has to give in compressing code into a space with an absolute limit of one thousand and twenty four characters. Certainly, neat presentation goes out the window, along with potentially meaningful names, stack comments, and even the title line. Naturally, one tries to retain as many of these as possible. More importantly, everything that is trivial or superfluous has to be stripped out mercilessly. This leaves only the core of the program, its essence. Divining the essence of a problem is the beginning of understanding.

Normally I would throw away the one-screen version once it was stable, and start coding a fuller solution afresh.

Comparison

Although prompted by the Curly Control Set, I will compare UCS to the ANSI control set, as I do not wish to do Giorgio any injustices by erroneously criticizing a wordset that I am not familiar with.

Syntax

A control structure starts with BEGIN and ends with either END or AGAIN. Within a structure, any number of WHENs may appear. WHEN must be paired with END or AGAIN. WHEN substructures may not be nested. BEGIN structures, however, may be nested. WHILE is functionally equivalent to 0= WHEN END, and should be treated as a WHEN pair for syntax purposes. The same considerations apply to UNTIL, which is equivalent to 0= WHEN AGAIN.

Semantics

BEGIN has no run-time action, it simply marks the beginning of a structure. The END that pairs up with BEGIN equally has no run-time action. AGAIN, whether paired with BEGIN or WHEN, gives an unconditional branch to just after BEGIN. END, when paired with WHEN, gives an unconditional branch to just after the final END or AGAIN. WHEN takes a flag. On false, it skips to just after the closing END or AGAIN. On true, execution continues sequentially. As stated above, WHILE has the same action as 0= WHEN END, but is more efficient. On false, it skips to just after the final END or AGAIN. Otherwise execution continues sequentially. Similarly, UNTIL is more efficient than 0= WHEN AGAIN. Execution continues just after BEGIN on false, and sequentially otherwise.

Usage

Usage is compared to the proposed ANSI control set on screens two and three of the accompanying listing. BEGIN AGAIN is the same with both ANSI and UCS. BEGIN AGAIN is one of two basic structures in UCS. The other is BEGIN END. BEGIN END has no equivalent in ANSI. It does not affect the flow of control at all and is, therefore, the structural equiva-
The UCS equivalent of the CASE statement is rather more general, and is better compared to the LISP COND or the Occam extended IF. The extra code in the illustration (the DUPS, etc.) shows what typically would be required to simulate an Eakor CASE.

On the final screen is what may be deemed advanced usage of the ANSI set. The first structure is more understandable in UCS, as the exit conditions (a) and (b) are positioned next to the decision to branch, rather than at the end of the structure in reverse order (d). It is, however, less efficient, as leaving via condition (a) incurs an overhead of one unconditional forward branch in UCS. Given that this only occurs with the first exit path, no matter how many WHENs are present, this is not too detrimental.

ANSI, on the other hand, starts to suffer as the number of WHILES increases, as one hops and skips out of terminating ELSE ... THEN ELSE ... THENs. It is perfectly possible to use WHILE ... ELSE in ANSI to avoid this but, as I have never seen the construct published, must assume that this is not typical usage. The 0-s are irrelevant, and are merely there to indicate that the logic of WHEN is reversed with respect to WHILE. (WHILE ELSE, in ANSI, would also demand reversed logic to WHILE.)

The ANSI rationale (at least the first draft proposal) states, “The use of more than one additional WHILE is possible but not common.” This is convenient, as two exits represents about the limit of legibility. This is illustrated with the final example, which is less than crystal clear in ANSI. (In case you were wondering, if the first WHILE succeeds, the section labeled (a) is executed and the structure exited. If the second WHILE succeeds, (b) is executed and the structure left. If the third, then section (c) runs and execution continues at the BEGINs. If none succeed, execution continues at the BEGINs.) This took some time to construct, whereas the UCS equivalent was trivial. UCS suffers no increase in complexity as the number of WHENs increases. Perhaps the use of more than one additional WHILE would be more common, if not for its complexity and unreadability.

ANSI is complete, in that it can be used to create any conceivable system of branches, but there comes a point when GOTOs would be more comprehensible. UCS, on the other hand, is not complete, but does deliver a useful subset without
Figure Four.

<table>
<thead>
<tr>
<th>Text</th>
<th>BEGIN</th>
<th>WHEN</th>
<th>END</th>
<th>AGAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>Save vars</td>
<td>Compile branch</td>
<td>Resolve WHEN forward branch</td>
<td>Back branch start history</td>
</tr>
<tr>
<td>Executing history</td>
<td>Restore vars</td>
<td>Resolve forward branch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Increasing complexity. It is not compatible with previous systems, but the effects of maintaining compatibility at any cost are amply illustrated by the development of the IBM PC. You pay your money and you take your choice.

Assumptions

In order to fit the code into one screen, certain assumptions have been made. It is assumed that the words BRANCH and ?BRANCH are present, and that they expect the following cell in the code space to contain an absolute address for them to branch to. Furthermore, it is assumed that the code and data space are contiguous, so that it is meaningful to use HERE and , (comma) to provide these branch addresses. Finally, it is assumed that the compilation stack is the data stack.

Stack Comments

Although I have been able to retain stack comments in the space available, they are rather terse and deserve some explanation. "a" represents an address and "e" an execution token. Where a word finishes with EXECUTE, the stack comment assumes that the EXECUTE word has no stack effect. The comments for IMMEDIATE word? show the compile-time stack effects only. At run time, the words BEGIN, END, and AGAIN have no stack effect, and WHEN, WHILE, and UNTIL absorb a flag.

Overloading

In order to reduce the number of structure words, AGAIN and END are overloaded, each having two distinct operations depending on context. This is achieved by using vectored execution. BEGIN sets the execution vectors "E and "A to the actions associated to closing a BEGIN, B-END, and B-AGAIN. WHEN sets them to W-END and W-AGAIN. When a WHEN is closed, "E and "A are reset—by W-END or W-AGAIN—to B-END and B-AGAIN, respectively. To allow for nesting, the contents of "E and "A are saved on the stack by BEGIN and are restored when the structure is complete.

Resolving Backward References

The address in the code being compiled when BEGIN is encountered is held in a variable B-H (BEGIN-HERE) so that it is accessible at all times, and backward references can be resolved when they are encountered. To allow for nesting, the contents of B-H are saved on the stack by BEGIN and are restored when the structure is complete.

Resolving Forward References

There are two types of forward references. The simpler is that created by WHEN. The address to be filled is left on top of the stack, and is resolved by the WHEN'S closing partner. When the partner is END, a forward reference of the second type is made. This cannot be resolved until the final END or AGAIN is reached. Each of these forward references is covered on the stack by an execution token, to form part of the executable history.

EXECUTABLE HISTORY

As a control structure is written into the code space, a program is built up on the stack, which will be executed when the control structure is completed. BEGIN lays down the first part of this program, which is called FINISH and will be the last part to be executed before control is handed back to the compiler. It has three data items associated with it, which are the original values of the three variables. FINISH restores these, allowing nesting to work. Above FINISH may come zero or more E-RES's, whose function is to resolve one unresolved forward reference each. E-RES forces execution of the stack program to continue, by ending with EXECUTE. The final END or AGAIN initiates execution by also ending with EXECUTE.
Compiler Security

Compiler security is not included in the code presented here, because of space considerations. Standard techniques can be used, and the syntax is sufficiently simple and rigid that all illegal constructs are readily detected. Overloading END and AGAIN reduces the number of possible illegal constructs.

Extensions

The use of the executable history technique means that any extension may be added without altering existing code. Certain constructs are poorly named and would benefit from synonyms. Unfortunately, IF (ELSE) THEN would have to be quite smart, and probably could not be written in terms of existing words. I have not attempted to incorporate counted loops, as I have certain opinions on this subject which would distract from the intent of this article. Best to let sleeping dogs lie, as the old saw goes.

Conclusion

At its most spartan, a powerful control set can be constructed out of four words (UNTIL and WHILE do not add any functionality to the word set). More importantly, the disparate control structures can be unified into a single adaptive structure. The prices to pay are non-compatibility and overloaded operators.

References

Kourtis Giorgio's Curly Control Set (brilliant name, reminds me of a British advertising slogan: "Watch out, they taste curly!") appears in Forth Dimensions, XIII/6 and XIV/1.

The FOR NEXT demonstration referred to in the second paragraph was made in Fortthwrite issue 47, in the article about Loopy, a minimal subset language. Fortthwrite is the FIG-UK chapter magazine.

The one-screen discipline was suggested by Mike Lake in Forth Dimensions, XIII/3.

Chris Hainsworth's Extended Case appears in Fortthwrite issue 40. This gave the basic structure of UCS.

The Universal Delimiter appears in Fortthwrite issue 53 and shows an extension of the techniques used in UCS.

The comparison with ANSI is based on Wil Baden's marvelous pieces in the FORML Proceedings, particularly '89 and '90.

My opinion of DO LOOP is expressed in more issues of Fortthwrite than I care to mention. Of particular relevance here is issue 47, which melds it into the Hainsworth structure; and issue 58, which proposes a radical refactoring.

Gordon Charlton is a part-time hobbyist programmer and full-time house spouse who migrated to Fortth from LISP and Pascal after a Turkish friend told him that Fortth was a weird language and that he would consequently like it. He was right. Gordon is also, probably, the world's only Loopy programmer. He is currently the Events and Meetings Secretary of FIG-UK, and contributes regularly to Fortthwrite. His last major project was a string-pattern matcher which was presented at EuroFORMIL '91 and '92. If anyone can provide a rigorous description of the Ratcliffe-Oberschelp algorithm, he would be pleased to hear from them.
One-Screen Unified Control Structure

\ One-Screen Unified Control Structure  G Charlton 27Sep92
variable 'E  variable 'A  variable B-H
: FINISH ( a a a) b-h ! 'e ! 'a ! ;
: E-RES ( e a) here swap ! execute ; : B-END ( e) execute ;
: B-AGAIN ( e) compile branch b-h @ , execute ;
: B-E,A [''] b-end 'e ! [''] b-again 'a ! ;
: W-END ( a-a e) compile branch here 0 , here rot !

\ Screen 2
\ One-Screen Unified Control Structure -- Usage
BEGIN ... AGAIN -> BEGIN ... AGAIN
BEGIN ... WHILE ... REPEAT -> BEGIN ... WHILE ... AGAIN
BEGIN ... UNTIL -> BEGIN ... UNTIL END
IF ... THEN -> BEGIN WHILE ... END
IF ... ELSE ... THEN -> BEGIN WHEN ... END ... END
CASE ...
... OF ... ENDOF dup ... = WHEN drop ... END
... OF ... ENDOF dup ... = WHEN drop ... END
... ENDCASE ... drop END

\ Screen 3
\ One-Screen Unified Control Structure -- Usage, continued
BEGIN ... WHILE -> BEGIN ... @= WHEN ... (a) END
... WHILE ...
... @= WHEN ... (b) END
...
REPEAT AGAIN
... (b) ELSE ... (a) THEN
BEGIN BEGIN ... WHILE -> BEGIN ... @= WHEN ... (a) END
... WHILE ...
... @= WHEN ... (b) END
... WHILE ...
... @= WHEN ... (c) AGAIN
...
REPEAT AGAIN
... (c) [ 2 ] SO REPEAT
... (b) ELSE ... (a) THEN
Charles Moore's Fireside Chat '92

As related by C.H. Ting

Chuck discussed the newly released 386 OK, its implementation and its philosophy. OK is the next incarnation of Forth. It has many of Forth's attributes, but is simpler and more powerful. It exists in code only, no source. The best way to deal with a computer is through its code—not the source, which is only a description of the code, not the code itself. "The map is not the territory; a description is not the program." Chuck also discussed his CAD implementation on OK and the general characteristics of the P21 chip, which is under development.

—Dr. C.H. Ting

OK is the future of Forth. It is what Forth should become. For 20 years, I tried to make Forth more readable and more compatible to other programming languages. Now, I give up. The problem is fundamental. All programming languages, including Forth, are text-based languages. The problem is intrinsic, in that the language is used to describe a program. A text-based language has problems in syntax, like infix notation... Forth has less trouble in this respect, but it is still a description, not the program itself.

Forth has the advantage that the source and object code are all accessible to the programmer. The programmer can express himself and modify his code quite freely. The industry is moving in the opposite direction. Intel goes out of its way to make it difficult for programmers to modify code in the code segment. There is a dialectic contrast here. Forth empowers the programmer, but the establishment wants constraints and control.

The problem in programming languages is the syntax of the underlying text. English description of a program is impossible, just as symbolic expression of mathematics is impossible. Goethe said that mathematical truth cannot be proven. Symbology cannot be the description. It is impossible to describe a program by text. A program is best expressed by the binary bits, but the binary bits have no intrinsic meaning. A program runs; it does what you want to control. Text, the description of the program, cannot do it.

In OK-CAD, you have all the code you need to deal with the task you have to do, and that's all there is to it. There is no source. The closest thing to the source code are the pages of notes I keep in a binder. The temptation to document the code is strong, but the value is nil.

OK is not text based. The map is not the territory. Description is not the real thing.

The industry is very much in virtual reality, in modeling and simulation. Boeing is very strong in modeling on computers. Mechanics would not agree that the models would work. It is a typical GIGO, garbage-in, garbage-out.

OK exists in code. It allows you to do whatever you want. Never mind how it came to being. I wrote it first using DEBUG... Most of the time I trust the code is there and I don't worry about it. OK is the incarnation of Forth in the '90s.

Forth is based on a virtual machine with two stacks. Phil Koopman, Jr. said in a recent paper that Forth is a way of factoring. Forth is kind of modular, and OK is very modular. I will give you a few examples. The names in OK are spelled funny. The most common symbols are the arrows: ^ (the up arrow), v (the down arrow), > (the right arrow), and < (the left arrow). I used them often and consistently. ^ always means increase, moving up, and so forth. Here are some words:

: ^ 1 CHANGE ;
: v -1 CHANGE ;
: > 100 CHANGE ;
: < -100 CHANGE ;

These fragments are used very often. In fact, I have 12 versions of them in OK and CAD. Since they are used so often, one might want to code a generic version which could be used everywhere. In a generic version, you may want to clip the value in the register, and do other things like range checking, etc. However, a universal version does not exist. Instead, I have a universal construct like this:

^  
\[ \]
\[ V \]
\[ > \]
\[ < \]
CHANGE ... AX +! ;

Using the 386 machine instructions, each invocation of CHANGE uses only three 386 instructions. Here, a high-level language is not helpful. These small pieces of code fragments are best done in machine code.

Most of the code is not in subroutines, it is in code fragments to be jumped to, not called. You jump to a piece of code. Eventually you jump to another menu, not return to some caller.
Look at all the computer applications. Most often you are presented with a screen, which gives you some choices. You scroll the screen, sometimes call other screens and use some keys to make the choices. The meaning of keys changes with the context. Giving each key a special name will get you into trouble, because after the context changes the keys will have completely different meanings. The context is the whole screen. You do not need to have a word displayed on the screen to tell you what the screen context is.

I use the 386 only as a historic instance. OK is really designed for P20. When P20 is available, it will have OK and eForth. OK is more intuitive to use. It is easy, and you can use it to explore P20. If we make thing easy to get into, machine language programming can be taught in grade school. I am persuaded enough to build it and use it in the last four years. You should carefully monitor what I am doing, and jump in, if you will, when you are ready.

The CAD system is now complete. I have not spent much time changing it. The last thing I added was design rule checking. I originally thought it was not necessary; the chip should be correct by design, not by checking. However, I have to do it to convince myself that the design does not have any problem. I thought about it for a long time, about a month, before I started coding. I had one page of notes scribbled on a piece of paper, and I spent a couple of hours coding it. Rule checking has been a hot topic in the IC industry. There are a number of algorithms. The one I chose was the one everybody else rejected, of course.

I kept a table of rectangles in memory. The layer of first metal is the most complicated, and it has 20,000 rectangles. I simply compare the rectangles one by one to see if any two of them get too close. The code is very short, but it takes a long time to run through 20,000 x 20,000 comparisons. It took half an hour to check the first metal layer. This is the longest program I ever ran on OK. The other layers are much simpler, and take about ten seconds to run through. The rule checker stops when an error is detected, and the screen shows the tiles around the erroneous rectangles, with the cursor sitting on one of the rectangles. I can correct the mistakes and run it over again.

The code of this design rule checker is only a few hundred bytes long. It is so small because of Forth. I indeed have a Forth system, well factored and easy to use.

**Questions from the Audience**

**Does OK have two stacks?**

OK has one stack for subroutine calls and for temporarily storing register contents. I have a virtual data stack in the 386 registers. The order is AX, BX, BP, and so forth. The registers are generally used in that order. My convention is that, in a subroutine, all the registers are assumed to be free to use. The caller is responsible for saving and restoring registers that might get changed. This practice is, again, contrary to other conventions. It was done just to irritate people.

**How did you implement OK on a 386?**

The 386 is a very complicated machine. It has more than 500 instructions. I keep a well-thumbed Intel 386 manual. P20 has only 28 instructions and I have memorized all of them. Most people would start with a cross-compiler. Implementing 386 OK, I started using DEBUG to enter the code until the menu system worked. Then I could modify the system and add new code by using OK itself. For major changes, I still use DEBUG.

Is OK an 'O' and a 'K' or is it 'Zero K'?  
Let me say a few words about the CAD system.

I did the chip layout in tiles. There are 600 x 600 tiles in the P20 design. Each tile has five layers, internally represented by a 32-bit word. The entities contained in a tile have different meanings depending on where they are. The meaning of the layout cannot be carried in words, but they are carried fully in the tiles.

Most CAD systems try to use symbolic description of the layout of a chip. I used it to lay out the pads around the core of the chip. I used it in ShBoom because I didn’t have enough memory to hold the pads. The symbolic description was terrible. I cannot move them easily, and I cannot align them to the coordinates I want, so that the pad can be connected correctly to the signal traces. Finally, I moved the design to 386 OK, which has more memory. All the pads were laid out in tiles and the problems disappeared. The best representation of a picture is the picture itself, not its description.

OK is distributed in code. How can other people contribute to OK?

OK takes 64 Kbytes. People can change it and build new applications in the 64K chunk. We can collect these images and distribute them on a single floppy disk.

Compatibility is a taboo, here. I have no intention to make OK system compatible. The code of OK 1.1 on Novix, OK 2.1 on ShBoom and OK 3.1 on 386 are all similar but not identical.

You have no source listings. How do you move OK to a new processor?

The most important structure in OK is the menus. The menu structure can be implemented on any processor, using different techniques. The details will change, but the menu structure will be the same. You can get much closer to a machine without a language. It is like music and the score. The score is not the music. Different musicians play the same score. Some will produce beautiful music, others will produce terrible music—even if they all play correctly according to the same score. There are lots of things the score does not tell about the music.

Getting back to P20, MuP21 will be out in another week, and OK 4.1 on P21 will be the ultimate OK. So far, the chip doesn’t work. However, I followed the sequence of evolution without the benefit of working silicon. The design has been changed and improved greatly since it was first conceived. It got simpler. An example is the master clock. I started with one clock, then it was necessary to have a second. The synchronization between the two clocks became a real problem. Now there is no clock. I am using an analog delay line to control the timing. The circuits are much simpler and more powerful.
The reasons the chip didn't work were many. The key suspect is the distance between rectangles. The design rules published by the manufacturers are not clear what the distance really means. Is it the absolute diagonal distance between corners of rectangles or the lateral distance between the edges of the rectangles? I have to move to a fail-safe direction, consistent with my understanding of the rules.

The chip area is 100 mil square. It is divided into 600 x 600 tiles, which I laid out one by one. The image of the layout is 1.5 Mb in size. I converted the tiles into rectangles, and saved the rectangles to a file. Interestingly, the rectangle file is also 1.5 Mb. The rectangle file is then ZIPPed down to 300 Kb.

The code of CAD is about 3 Kb. Very small, compared to industry standards. The risk of software is that it becomes bigger and more complicated. Simpler software is always more reliable. 700,000 lines of code cannot be reliable. You cannot check them all. Simpler software means that you can check it completely... I read that the problem of the Patriot missile was traced to its floating-point calculation. In a long sequence of calculations to follow the trajectory of the target missile, the truncation errors in the floating-point calculation made the missile unworkable. I was a great believer in software until I learned hardware.

In developing the P20 chip, I rely heavily on my simulator. I want to trust my simulator, but it is not yet proven. If I get the simulator proven, I will be able to move on to design other chips. I have lots of chips lined up in the pipeline to be designed.

How do you decompile the machine code?
P20 code can decompile very easily into Forth. However, there is no room for comments. One important clue is the names, which one can assign to any location in the memory. The decompiler in 386 OK is not yet complete. It groups the bytes in a 386 instruction and displays them in one line. The instruction and its arguments are not translated to their Intel mnemonics.

If a map is not the territory, is it necessary to have a map?
A map is useful, but it is different from the territory. When the difference is subtle, you may confuse yourself.

I am working on P32 and P24. In P21, the last bit was added to take care of the carry in the ALU operations and it is also used to control memory addressing, to differentiate DRAM from SRAM. The clocks are done in analog delay lines. The T and N registers are tied to the ALU. You just enable the output and the results are latched back into T. It takes 10 ns from enabling of address input, through the ALU, to get the data to the output pins.

Silicon design is very challenging. A NAND gate has many inputs and an output. If you want more driving current from a NAND gate, you may use another output driver and then invert the driver. Or you can invert the inputs. Or you can invert the output. There are so many different approaches, and it becomes a holistic problem. I don't know how to come to an optimal implementation. Short of doing the experiments yourself and get experience the hard way, you don't have a note on how to get it done right.

In the meantime, OK is now out and I hope you will have fun with it. Thank you.

Sincerely,
Guy M. Kelly
2507 Caminito La Paz
La Jolla, California 92037

Kelly's Comparisons Clarified

Dear Marlin,

There may be some confusion about the meaning of the timing information contained in Tables Two and Three of my article, "Forth Systems Comparisons" (FD XIII/6), as evidenced by comments one and two in the much-appreciated letter by Don Kenney in FD XIV/3.

The comments accompanying the benchmark code were meant to illustrate that the empty loop times were subtracted from all the other raw times (except for the Sieve). Hence, for instance, the empty loop time for riFORTH is shown as greater than the threading time because it has already been subtracted from the raw time.

The probable reason that riFORTH did not perform better in the Sieve benchmark is that the version tested used a high-level DO LOOP construct, and the loop times were not subtracted from the Sieve times.

I would have to disagree with Mr. Kenney that hand calculating the times would have been easier, considering the number of versions of Forth tested, the number of words and tests per Forth, and the fact that the timing differences were in the same ball park (approximately ten to 30 percent) that the hard timings were in error (reported as four and seven percent).

My thanks to all those who took the time to comment on the article, it made it all worthwhile!

Sincerely,
Guy M. Kelly
2507 Caminito La Paz
La Jolla, California 92037

(Foreword, continued from page six.)

I was glad to see the FIG presence, but without other Forth vendors I question if many conventioneers got the Forth message. Too bad there weren't some controller-operated, fun gizmos, like the traveling display at the Anaheim (1988) programming contest, to show that Forth actually works. It could give some Forth vendors a chance to show off their hardware. If the gizmos are made transportable, they could be made available to FIG displays at other conventions. And demos of Forth programming (there's a lot of postfix paranoia) would let the curious actually see some of the Forth systems available. Forth definitely needs more marketing pizzazz.

Someone at the FIG booth demonstrated a clever method of creating and then downloading headerless Forth code through a serial port to the controller. Perhaps he would describe it further in an article?

Motorola won the "Chutzpah" award in marketing, hands down. For the Oktoberfest beer bust at the convention, Intel handed out fancy glass mugs imprinted with their logo. Motorola then passed out blue insulating blankets that wrapped around the mug, covering Intel's logo with their own. I presume a suitable revenge is being planned...

Yours truly,
Walter J. Rottenkolber
P.O. Box 1705
Mariposa, California 95338
By now you know that HS/FORTH gives you more speed, power, flexibility and functionality than any other language or implementation. After all, the majority of the past several years of articles in Forth Dimensions has been on features found in HS/FORTH, often by known customers. And the major applications discussed had to be converted to HS/FORTH after their original dialects ran out of steam. Even the public domain versions are adopting HS/FORTH like architectures. Isn't it time you tapped into the source as well? Why wait for second hand versions when the original inspiration is more complete and available sooner.

Well, it was a dirty job, but we finally had to do it. Now you can run lots of copies of HS/FORTH from Microsoft Windows in text and/or graphics windows with various icons and pif files available for each. Talk about THE tool for hacking Windows! But, face it, what really like is cranking up the font size so I can still see the characters no matter how late it is. Now that’s useful.

Good news, we’ve redone our DOCUMENTATION! The big new fonts look really nice and the reorganization makes all that functionality so much easier to find. Thanks to excellent documentation, all this awesome power is now relatively easy to learn and to use.

Naturally we continue tweaking and improving the internals, but by now the system is so well tuned that these changes are not individually of any significance. They just continue to improve performance a bit at a time, and enhance error detection and recovery. Update to Revision 5.0, including new documentation, from all 4.xx revisions is $99. and from really old systems the update is $149.

And since Spring is coming, IT IS TIME FOR OUR SPRING SALE. Thru the end of May you get to pick two extra utility packages free for each Professional or Production Level system purchased, or get a free Online Glossary with help file utility with each Personal Level system purchased.
Numbers

C.H. Ting
San Mateo, California

In this lesson, we shall discuss the way Forth handles integers. Integers are numbers from -32768 to 32767. This range of numbers is most convenient to be stored and processed in Forth. It is very surprising that many real-world problems can be represented and solved using numbers in this range. Forth can handle larger numbers, and even floating-point numbers, but these are topics outside the scope of this lesson.

**Example One. Money Exchange**

The first example we will use to demonstrate how numbers are used in Forth is a money exchange program, which converts money represented in different currencies. Let's start with the following currency exchange table:

<table>
<thead>
<tr>
<th>Currency</th>
<th>Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT</td>
<td>24.55</td>
</tr>
<tr>
<td>HK</td>
<td>7.73</td>
</tr>
<tr>
<td>RMB</td>
<td>5.47</td>
</tr>
<tr>
<td>Silver</td>
<td>1 Ounce</td>
</tr>
<tr>
<td>Gold</td>
<td>1 Ounce</td>
</tr>
<tr>
<td>NT</td>
<td>1 Dollar</td>
</tr>
<tr>
<td>HK</td>
<td>1 Dollar</td>
</tr>
<tr>
<td>RMB</td>
<td>1 Dollar</td>
</tr>
<tr>
<td>Silver</td>
<td>356 Dollars</td>
</tr>
<tr>
<td>Gold</td>
<td>4.01 Dollars</td>
</tr>
</tbody>
</table>

We shall use the U.S. dollar as the standard currency, and convert all other currencies to dollars first. All arithmetic operations will be carried out in dollars. The dollars can then be converted back to any other currency.

We define words to convert other currencies to the dollar by using the names of the corresponding currencies. To convert from dollars to another currency, the word is preceded by the $ sign.

```forth
: NT ( nNT -- $ ) 10 245 */ .s ;
: NT  ( $ -- nNT ) 245 10 */ .s ;
: RMB ( nRMB -- $ ) 100 547 */ .s ;
: $jmp ( $ -- nJmp ) 547 100 */ .s ;
: HK  ( nHK -- $ ) 100 773 */ .s ;
: HK  ( $ -- $ ) 773 100 */ .s ;
: gold ( nOunce -- $ ) 356 */ .s ;
: gold ( $ -- nOunce ) 356 / .s ;
: silver ( nOunce -- $ ) 401 100 */ .s ;
: silver ( $ -- nOunce ) 100 401 */ .s ;
: ounce ( n -- n, used to improve syntax ) ;
: dollars ( n -- ) . ;
```

With this set of money exchange words, we can do some tests:

5 ounce gold .
10 ounce silver .
100 SNT .
20 $RMB .

If you have many different currencies in your wallet, you can add them in dollars:

1000 NT 500 HK +
320 RMB +
dollars ( prints total worth in dollars )

If I am in Hong Kong at the time, the total amount can be readily converted to Hong Kong dollars:

1000 NT 500 HK + 320 RMB +
$HK dollars
( converts to Hong Kong dollars and prints it )

**Exercise One. A business trip.**

Now we have a fairly powerful money exchange computer with us. Suppose you depart San Francisco with 1000 dollars in your pocket. You go to Hong Kong and buy a VCR with 1200 HK. Go to Shanghai and sell it for 2000 RMB. Then come back to Hong Kong and spend 900 HK for fun. Go to Taipei and buy a portable PC with 30000 NT. How much money in U.S. dollars do you have remaining?

The answer typed backwards is:

```forth
srallod - TN 00003 - HK 009 + BMR 0002 - KH 0021 0001
```

Try it.

**Exercise Two. Temperature conversion.**

Converting temperature readings between Celsius and Fahrenheit is also an interesting problem. The difference between temperature conversion and money exchange is that the two temperature scales have an offset in addition to the scaling factor.

```forth
: F>C ( nFahrenheit -- nCelcius )
  32 -
  10 18 */
  ;

: C>F ( nCelcius -- nFahrenheit )
  18 10 */
  32 +
  ;
```

90 F>C . shows the temperature on a hot summer day and
0 C>F . shows the temperature in the cold winter.

In the above examples, we use the following Forth arithmetic operators:
Forth Dimensions

1. Uses numbers pops the required numbers from the stack. The most accessible number is on the top of the stack, like the names.

2. One must notice that the order of the two numbers used by the operators is immaterial, while the order is important for the operators yet to be executed. Operators take parameters from the stack and leave results there for subsequent operators to use. A program can be built easily by stringing together subroutines. The subroutines can call other subroutines, and so on. The subroutines are Forth operators, and can be nested almost indefinitely. This is a very important reason why Forth is simple in its architecture and also in its syntactical structure.

3. However, it happens very often that the order of the numbers on the stack is not correct for an operator which needs them, like – and /. There is a set of stack operators to rearrange numbers on the stack. The five most important, classic stack operators are:

   - **DUP** \((n - n n)\) Duplicate the top of stack.
   - **SWAP** \((n1 n2 - n2 n1)\) Exchange top two numbers on stack.
   - **OVER** \((n1 n2 - n1 n2 n1)\) Duplicate the second number on stack.
   - **ROT** \((n1 n2 n3 - n2 n3 n1)\) Rotate third number to the top of stack.
   - **DROP** \((n -)\) Discard the top of stack.

4. **Example Three. Rectangles.**

   A rectangle is specified by the \((x, y)\) coordinates of its upper-left and lower-right corners. With these four integers on the stack, we can compute the area, the center, and the perimeter of a rectangle:

   ```forth
   : area ( x1 y1 x2 y2 -- area )
   ROT - ( x1 y2 y2-y1 )
   SWAP ROT - ( y2-y1 x2-x1 )
   * ( area ) ;
   : center ( x1 y1 x2 y2 -- x3 y3 )
   ROT 2/ ( x1 x2 y3 )
   SWAP ROT 2/ ( y3 x3 )
   SWAP ( x3 y3 ) ;
   : sides ( x1 y1 x2 y2 -- sides )
   ROT - ABS ( x1 x2 y2-y1 )
   SWAP ROT - ABS ( y2-y1 x2-x1 )
   + ( sides ) ;
   ``

5. **Logic Operators**

   Computers use logic operators to determine and follow different execution paths. Logic operators themselves are very simple and easy to understand. However, the combination of many levels of logic operations, and the multitude of different pathways in a large program, makes the computer
seem very complicated, even to the point of showing some intelligence.

Here we introduce some of the logic operators associated with numbers, and the branching operators which use the results of logic operators to select different operations.

Forth uses numbers to represent logic levels. There are only two logic levels, true and false. True is represented by any number which is not zero (usually a -1), and false is represented by zero. The number representing logic levels is often called a flag.

\[
\begin{align*}
> (\text{n1 n2}) & \quad \text{Return true if n1>n2.} \\
< (\text{n1 n2}) & \quad \text{Return true if n1<n2.} \\
= (\text{n1 n2}) & \quad \text{Return true if n1=n2.} \\
0= (\text{n}) & \quad \text{Return true if n=0.} \\
0< (\text{n}) & \quad \text{Return true if n<0.} \\
\text{NOT (f)} & \quad \text{Return true if f is false.} \\
\end{align*}
\]

A flag can be used to select one of the two execution paths by the following constructs inside a colon definition:

\[ (f) \text{ IF <true clause> ELSE <false clause> THEN} \]

Example Four. Weather Reporting.

The following colon definition illustrates the use of logic and the branch:

\[
\begin{align*}
\text{weather (nFarenheit --) } \\
\text{DUP 85 >} \\
\text{IF ." Too hot!" DROP} \\
\text{ELSE 55 <} \\
\text{IF ." Too cold."} \\
\text{ELSE ." About right."} \\
\text{THEN} \\
\text{THEN;} \\
\end{align*}
\]

You can type the following commands and get some responses from the computer:

90 weather Too hot!
70 weather About right.
32 weather Too cold.

Loop Operators

We shall be concerned now with only the definite loop operators used in the following format in a colon definition:

\[ (\text{nLimit nIndex}) \text{ DO <repeat-clause> LOOP} \]

DO takes two parameters off the stack. The top number is the starting index of the loop and the second number is the upper limit of the loop index. After entering the loop, the repeat clause is repeatedly executed. LOOP increments the loop index from nIndex to nLimit. When the index is equal to nLimit, the loop is terminated. In the repeat clause, a special operator I returns the current loop index on the stack.

A simple example of the loop structure follows:

Example Five. Print the multiplication table.

\[
\begin{align*}
\text{OneRow (nRow --)} \\
\text{CR} \\
\text{DUP 3 .R 3 SPACES} \\
13 1 \\
\text{DO I OVER *} \\
4 .R \\
\text{LOOP} \\
\text{DROP ;} \\
\end{align*}
\]

\[
\begin{align*}
\text{Table (--)} \\
\text{CR CR 6 SPACES} \\
13 1 \\
\text{DO I 4 .R LOOP (display column numbers)} \\
13 1 \\
\text{DO I OneRow LOOP} \\
\end{align*}
\]

Typing TABLE will cause the multiplication table to be displayed in a neat format.

With these new Forth operators, we can now write a fairly substantial program, using many of the operators to demonstrate how they are combined to do useful work.

Dr. C.H. Ting is a noted Forth authority who has made many significant contributions to Forth and the Forth Interest Group. His tutorial series will continue in succeeding issues of Forth Dimensions.
Optimizing in BSR/JSR-Threaded Forth

Charles Curley
Gillette, Wyoming

The purpose of this paper is to describe a code optimizer for a 68000-based JSWBSR-threaded Forth compiler. The code operates in the traditional Forth single-pass compiler, optimizing on the fly. The result includes words which execute in fewer instructions than the words called out in the source code.

Historical Note
The Forth used for the code described herein is FastForth, a full 32-bit BSR/JSR-threaded Forth for the 68000, described in unmitigated detail in "Optimization Considerations" (Forth Dimensions XIV/5). It is a direct modification of an indirect-threaded Forth, real-Forth. This is, in turn, a direct descendent of fig-Forth. (Remember fig-Forth?) This vocabulary, word names, and other features have been retained.

For those not familiar with 32-bit Forths, the memory operators with the prefix w operate on word, or 16-bit, memory locations. FastForth uses the operators F@ and F! for 32-bit memory operations where the address is known to be an even address. To avoid odd-address faults, the regular

This optimizer is a complete unit, and is dependent only upon the nature of the target processor

Forth operators @ and ! use byte operations.

The assembler used to illustrate is descended from the fig 68000 assembler by Dr. Kenneth Mantei. It is a typical Forth reverse Polish notation assembler. Typical syntax is: source, destination, opcode. The addressing modes relevant to the paper are as follows:

<table>
<thead>
<tr>
<th></th>
<th>Address register indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Address register indirect with post-increment</td>
</tr>
<tr>
<td>!</td>
<td>Address register indirect with pre-decrement</td>
</tr>
<tr>
<td>&amp;</td>
<td>Register indirect with a word of displacement</td>
</tr>
<tr>
<td>@#L</td>
<td>Absolute long address</td>
</tr>
<tr>
<td>#</td>
<td>Immediate data, word</td>
</tr>
<tr>
<td>#L</td>
<td>Immediate data, long</td>
</tr>
</tbody>
</table>

There is nothing particularly new conceptually here. Chuck Moore's cmForth includes an optimizer for the Novix NC-4000. The present paper describes an optimizer for a more traditional CISC instruction set, the Motorola 68000.

The Compiler
The compiler used in FastForth looks very much like a traditional indirect-threaded Forth. However, it lays down opcodes which call (via BSR or JSR instructions) lower-level words, rather than a list of addresses for NEXT to interpret. For example, the traditional word L is defined as follows:

```
: L SCR F@ LIST ;
```

In an indirect-threaded, 32-bit Forth, the compiler would build the header for L. This would be followed by a four-byte address for the code to be executed to interpret the word. The code field address is followed by a four-byte address for each of the three words called out in the source. This would be followed by the address of the exit code, laid down by the compiler directive ;.

In a BSR/JSR-threaded Forth, the compiler lays down BSRs ("branch to subroutine") or JSRs (jump to subroutine), as appropriate, to the words called out. The return code consists of an RTS instruction. The result may or may not be smaller than the indirect-threaded version, but it certainly will be faster. Whether the result is smaller or not depends on the mix of short BSRs (two bytes), long BSRs (four bytes), and JSRs (six bytes) laid down at compile time.

One optimization discussed in "Optimization Considerations" is to examine the last instruction of a word. If it is a BSR or JSR, that instruction can be twiddled to produce a BRA or JMP instruction.

Another optimization is to lay down in-line code instead of calls. This is particularly beneficial when calling short words (e.g., F@) from a distance, which would require a JSR instruction. Not only does the technique save run time (by eliminating a call and an RTS instruction), but it may reduce the size of words. One circumstance where this technique does not save space is where a four-byte word is copied in line to a location which would have required a short (two-byte) BSR.
Variables, constants, and user variables in FastForth are immediate words which compile in-line code, often a six-byte reference.

With these optimizations, the compiler produces the above for the sample word L, given earlier.

The total is sixteen bytes at most, compared to a firm twenty bytes for the indirect-threaded version. So the JSR/BSR version may be smaller, but certainly will be much faster!

Two more typical Forth optimizations are common and won’t be discussed very much.

If a phrase shows up a lot in a Forth program, it is common practice for the programmer to consolidate that phrase in a word with a meaningful name. This is optimizing for readability and dictionary size, rather than speed.

The second is to reduce words from high level to assembler. This requires the active intervention of the programmer, and the results are well worth it in terms of speed, and often worthwhile in terms of space. Alas, such optimizations only improve readability for those who know the relevant assembly language (and, sometimes, the relevant assembler), leaving the code more opaque to those without such skills.

The Optimizer Design
An optimizer for in-line code should be a single-pass optimizer, to be consistent with Forth’s traditional single-pass compiler. This would make difficult, for example, replacing long forward branches with short ones on the fly, but would result in much simpler code in the compiler. It must, then, operate at compile time, and so must consist of immediate words.

The optimizer should be an add-on, so that the user can add to the optimizer if he wishes to. However, it should also be carried over to the cross-compiler so as to produce a very efficient nucleus.

The optimizer should work silently, as far as the user is concerned. That is, it should require no changes in source code to be useful. This requirement separates out the optimizing compiler from the two common optimizing methods described above.

Initially, the optimizer word could be developed as a series of discrete words, but these would be replaced by one or more defining words and their daughter words.

One key to single-pass optimization during compilation is to have immediate words which examine previously compiled opcodes, and twiddle certain ones to produce tighter code.

Another key is to look for certain phrases which can easily be detected and easily twiddled. We could look for the phrase BLK F8, which shows up all over the nucleus, and optimize that. Instead, let us generalize, and look for phrases of the general type:

<USER VARIABLE> F8.

To detail this example: a user variable is an immediate word, which lays down the following phrase:

<offset> & | U AR0 MOV, AR0 S - [ MOV,

Translated into English, the first instruction moves data from a user variable indicated by an offset from the user area register to address register 0. The second pushes it out onto the stack. The result, examined in memory as word data, looks like:

| 41ee | <offset> | 2708 |

If the next word in the source code is F8 and it is immediate, it can look at here-less-six for the opcode 41ee. Finding it, it can twiddle the dictionary to produce the following code:

<offset> & | U AR0 MOV, AR0 S - [ MOV,

This produces the following memory dump:

| 41ee | <offset> | 2710 |

The first instruction still moves the contents of the user variable into the address register, but the second instruction now reads data from the location pointed to by the register, and pushes it onto the data stack.

The phrase <USER VARIABLE> F8 is now executed in two instructions instead of the previous four, and occupies six bytes instead of the previous ten. And the optimizer works for all user variables, even ones not defined at the time the optimizer is compiled.

Other two-word phrases were similarly identified and optimized, and some three-word phrases were also identified and optimized. As each phrase was identified, a defining word was built up, consisting of nested IF ... ELSE ... THEN clauses. The resultant words are monsters, and must be thoroughly understood by the programmer who seeks to modify them. In these two respects, they are un-Forthish, but the gain obtained by using them is worth the price.

These words must all be state smart. As they will run either at run time or at compile time, they must examine STATE and act accordingly. The action at run time is, of course, to execute their namesakes. Hence, in the run-time portion of the defining words, the phrase STATE F8 IF ... ELSE EXECUTE THEN.

In order for that phrase to work correctly, we must have the run-time address of the namesake in the dictionary. We require the namesake to be explicitly stated: ' it and comma
FIG MAIL ORDER FORM

HOW TO USE THIS FORM: Please enter your order on the back page of this form and send with your payment to the Forth Interest Group.
All items have one price and a weight marked with # sign. Enter weight on order form and calculate shipping based on location and delivery method.

"Were Sure You Wanted To Know..."

Forth Dimensions, Article Reference
151 - $4 0#
An index of Forth articles, by keyword, from Forth Dimensions

Forth Dimensions Back Volumes
A volume consists of the six issues from the volume year (May–April)

Volume 1 Forth Dimensions (1979–80) 101 - $15 1#
Introduction to FIG, threaded code, TO variables, fig-Forth.

Volume 3 Forth Dimensions (1981–82)
Forth-79 Standard, Stack Hacking, memory management, stack formats, timing stack, BASIC compiler, recursors and assembler.

Volume 6 Forth Dimensions (1984–85) 106 - $15 2#
Interactive editors, anonymous variables, list handling, integer solutions, control structures, debugging techniques, recursion, semaphores, simple I/O words, Quicksort, high-level packet communications, China FORML.

Volume 7 Forth Dimensions (1985–86) 107 - $20 2#
Generic sort, Forth spreadsheet, control structures, pseudo-interrupts, number editing, Atari Forth, pretty printing, code modules, universal stack word, polynomial evaluation, F83 strings.

Volume 8 Forth Dimensions (1986–87) 108 - $20 2#
Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS-Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

Integrate driven serial input, data-base functions, TI 99/4A, XMODEM, on-line documentation, dual-CFAs, random numbers, arrays, file query, Batcher’s sort, screenless Forth, classes in Forth, Bresenham line-drawing algorithm, unsigned division, DOS file I/O.

Volume 10 Forth Dimensions (1988–89) 110 - $20 2#
Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS-Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

Volume 11 Forth Dimensions (1989–90) 111 - $20 2#
Local variables, graphic filing algorithms, 80286 extended memory, expert systems, quaternion rotation calculation, multiprocessor Forth, double-entry bookkeeping, binary table search, phase-angle differential analyzer, sort contest.

Volume 12 Forth Dimensions (1990–91) 112 - $20 2#
Floored division, stack variables, embedded control, Atari Forth, optimizing compiler, dynamic memory allocation, smart RAM, extended-precision math, interrupt handling, neural nets, Soviet Forth, arrays, metacompletion.

FORTH DIMENSIONS CONFERENCE PROCEEDINGS

FORTH Dimensions Conference Proceedings

1980 FORML Conference Proceedings 310 - $30 2#
Address binding, dynamic memory allocation, local variables, concurrency, binary absolute & relocatable loader, LISP, how to manage Forth projects, n-level file system, documenting Forth, Forth structures, Forth strings. 251 pgs

1981 FORML Conference Proceedings 311 - $45 4#
Object-oriented Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectoried data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic debugging language formatting, NEXOS—a ROM-based multitasking operating system. 655 pgs

1982 FORML Conference Proceedings 312 - $30 4#
Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE loop words, number input wordset, I/O vectoring, recursive data structures, programable-logic compiler. 295 pgs

1983 FORML Conference Proceedings 313 - $30 2#
Non-Von Neumann machines, Forth instruction set, Chinese Forth, F83, compiler & interpreter co-routines, log & exponential function, rational arithmetic, transcendental functions in variable-precision Forth, portable file system, interface, Forth coding conventions, expert systems. 352 pgs

1984 FORML Conference Proceedings 314 - $30 2#
Forth expert systems, consequence reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, 1IP71B Forth, NEXOS—object-oriented programming, decompiler design, arrays and stack variables. 378 pgs

1986 FORML Conference Proceedings 316 - $30 2#
Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple inheritance system, automatic programming environment. 323 pgs

1987 FORML Conference Proceedings 317 - $40 2#
Includes papers from '87 euroFORML Conference. 32-bit Forth, neural networks, control structures, AI, optimizing compilers, hypertext, field and record structures, CAD command language, object-oriented lists, trainable neural nets, expert system. 492 pgs

1988 FORML Conference Proceedings 318 - $40 2#
Includes 1988 Australian FORML, human interfaces, simple robotics kernel, MODUL Forth, parallel processing, programmable controllers, Prolog, simulations, language topics, hardware, Wil’s workings & Ting’s philosophy, Forth hardware applications, ANS Forth session, future of Forth in AI applications. 310 pgs

1989 FORML Conference Proceedings 319 - $40 2#
Includes papers from '89 euroFORML. Pascal to Forth, extendable optimizer for compiling, 3D modeling with object-oriented Forth, C/C++ polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RX recompiler for on-line maintenance, modules, trainable neural nets. 453 pgs

1990 FORML Conference Proceedings 320 - $40 3#
Forth in industry, communications monitor, 6802 development, 3-key keyboard, documentation techniques, object-oriented programming, simplest Forth decomplier, error recovery, stack operations, process control event management, control structure analysis, systems design course, group theory using Forth. 441 pgs

Fax your orders 510-535-1295

* - These are your most up-to-date indexes for back issues of Forth Dimensions and the FORML proceedings.
BOOKS ABOUT FORTH

ALL ABOUT FORTH, 3rd ed., June 1993, Glen B. Haydon
201 - $90 4#
Annotated glossary of most FORTH words in common usage, including FORTH-79, FORTH-83, F-PC, MVP-FORTH. Implementation examples in high-level FORTH and/or 8086/88 assembler. Useful commentary given for each entry. 506 pgs

THE COMPLETE FORTH, Alan Winfield
210 - $14 1#
A comprehensive introduction, including problems with answers (FORTH-79). 131 pgs

eFORTH IMPLEMENTATION GUIDE, C.H. Ting
215 - $25 1#
eFORTH is the name of a FORTH model designed to be portable to a large number of the newer, more powerful processors available now and becoming available in the near future. 54 pgs (w/disk)

F83 SOURCE, Henry Laxen & Michael Perry
217 - $20 2#
A complete listing of F83, including source and shadow screens. Includes introduction on getting started. 208 pgs

FORTH: A TEXT AND REFERENCE
Mahlon G. Kelly & Nicholas Spies
219 - $31 2#
A textbook approach to FORTH, with comprehensive references

THE FIRST COURSE, C.H. Ting
223 - $25 1#
This tutorial is an introduction to the very minimum set of FORTH instructions so that you can start to use FORTH to solve practical problems in the shortest possible time. ... This tutorial was developed to complement the FORTH Course which skims too fast on the elementary FORTH instructions and dives too quickly in the advanced topics in a upper level college microcomputer laboratory. ... A running F-PC FORTH system would be very useful. 44 pgs

THE FORTH COURSE, Richard E. Haskell
225 - $25 1#
This set of 11 lessons, called The FORTH Course, is designed to make it easy for you to learn FORTH. The material was developed over several years of teaching FORTH as part of a senior/graduate course in design of embedded software computer systems at Oakland University in Rochester, Michigan. 156 pgs (w/disk)

FORTH ENCYCLOPEDIA, Mitch Derek & Linda Baker
220 - $30 2#
A detailed look at each f:forth instruction. 327 pgs

FORTH NOTEBOOK, Dr. C.H. Ting
232 - $25 2#
Good examples and applications. Great learning aid. poly-FORTH is the dialect used. Some conversion advice is included. Code is well documented. 286 pgs

FORTH NOTEBOOK II, Dr. C.H. Ting
232a - $25 2#
Collection of research papers on various topics, such as image processing, parallel processing, and miscellaneous applications. 237 pgs

F-PC USERS MANUAL (2nd ed., V3.5)
350 - $20 1#
Users manual to the public-domain FORTH system optimized for IBM PC/XT/AT computers. A fast, fast system with many tools. 143 pgs

F-PC TECHNICAL REFERENCE MANUAL
351 - $30 2#
A must if you need to know the inner workings of F-PC. 269 pgs

INSIDE F-83, Dr. C.H. Ting
235 - $25 2#
Invaluable for those using F-83. 226 pgs

LIBRARY OF FORTH ROUTINES AND UTILITIES
James D. Terry
237 - $23 2#
Comprehensive collection of professional quality computer code for FORTH; offers routines that can be put to use in almost any FORTH application, including expert systems and natural language interfaces. 374 pgs

OBJECT ORIENTED FORTH, Dick Foust
242 - $35 1#
Implementation of data structures. First book to make object-oriented programming available to users of even very small home computers. 118 pgs

SEEING FORTH, Jack Woehr
243 - $25 1#
"... I would like to share a few observations on Forth and computer science. That is the purpose of this monograph. It is offered in the hope that it will broaden slightly the stream of Forth literature..."
95 pgs

SCIENTIFIC FORTH, Julian V. Noble
250 - $50 2#
Scientific Forth extends the FORTH kernel in the direction of scientific problem solving. It illustrates advanced Forth programming techniques with non-trivial applications: computer algebra, roots of equations, differential equations, function minimization, functional representation of data (FFT, polynomials), linear equations and matrices, numerical integration/Monte Carlo methods, high-speed real and complex floating-point arithmetic. 300 pgs (Includes disk with programs and several utilities), IBM

STACK COMPUTERS, THE NEW WAVE
Philip J. Koopman, Jr. (hardcover only)
245 - $50 2#
Presents an alternative to Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC) by showing the strengths and weaknesses of stack machines (hardcover only).

STARTING FORTH (2nd ed.), Leo Brodie
245 - $25 2#
In this edition of Starting FORTH— the most popular and complete introduction to Forth— syntax has been expanded to include the FORTH-83 Standard. 346 pgs

WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++,
Norman Smith
270 - $15 1#
This book is an application language. More specifically, it is about how to write your own custom application language. The book contains the tools necessary to begin the process and a complete sample language implementation. [Guess what language!]
Includes disk with complete source. 108 pgs

ACM - SIGFORTH
The ACM SIGFORTH Newsletter is published quarterly by the Association of Computing Machinery, Inc. SIGFORTH's focus is on the development and refinement of concepts, methods, and techniques needed by Forth professionals.

Volume 1
Spring 1989, Summer 1989, #3, #4
F-PC, glossary utility, euroFORTH, FORTH '89 Workshop summary (real-time software engineering), Intel 80x86. Metacompile in cmForth, FORTH exception handling, string case statement for UF/FORTH. 1802 simulator, tutorial on multiple threaded vocabularies. Stack frames, duals; an alternative to variables, PocketForth.

Volume 2
#1, #2, #3, #4

Volume 3
#1 Summer '91
Co-routines and recursion for tree balancing, convenient number handling.

Volume 3, #2 Fall '91

1989 SIGForth Workshop Proceedings
Software engineering, multitasking, interrupt-driven systems, object-oriented FORTH, error recovery and control, virtual memory support, signal processing. 127 pgs

1990-91 SIGForth Workshop Proceedings
Teaching computer algebra, stack-based hardware, reconfigurable processors, real-time operating systems, embedded control, marketing FORTH, development systems, in-flight monitoring, multi-processors, neural nets, security control, user interface, algorithms. 124 pgs

For faster service, fax your orders 510-535-1295
DISKS: Contributions from the Forth Community

The "Contributions from the Forth Community" disk library contains author-submitted donations, generally including source, for a variety of computers & disk formats. Each file is determined by the author as public domain, shareware, or use with some restrictions. This library does not contain "For Sale" applications. To submit your own contributions, send them to the FIG Publications Committee.

Prices: Each item below comes on one or more disks any disk ≤ 1 #.

FLOTAt4th.Blk V1.4 Robert L. Smith C001 - $8
Software floating-point for fig., poly., 79-Std., 83-Std.
Forths. IEEE short 32-bit, four standard functions, square root and log.
*** IBM, 190Kb, F83

Games In Forth
Misc. games, Go, TETRA. Life... Source.
* IBM, 760Kb

A Forth Spreadsheet, Craig Lindley C003 - $6
This model spreadsheet first appeared in Forth Dimensions VII, 1-2. Those issues contain docs & source.
* IBM, 100Kb

Automatic Structure Charts, Kim Harris C004 - $8
Tools for analysis of large Forth programs, first presented at FORML conference. Full source; docs incl. in 1985 FORML Proceedings.
** IBM, 114Kb

A Simple Inference Engine, Martin Tracy C005 - $8
Based on inf. engine in Winston & Horn's book on LISP, takes you from pattern variables to complete unification algorithm, with running commentary on Forth philosophy & style. Incl. source.
** IBM, 162 Kb

Math Box, Nathaniel Grossman C006 - $10
*** IBM, 118 Kb

AstroForth & AstroOKO Demos, J.R. Aguirnissian C007 - $6
AstroForth is the 83-Std. Russian version of Forth. Incl. window interface, full-screen editor, dynamic assembler & a great demo. AstroOKO, an astronavigation system in AstroForth, calculates sky position of several objects from different earth positions. Demos only.
*** IBM, 790 Kb

Forth List Haeber, Martin Tracy C008 - $8
List primitive adds Forth to provide a flexible, high-speed environment for AI. Incl. ELISA and Winston & Horn's micro-LISP as examples. Incl. docs & source.
*** IBM, 170 Kb

8051 Embedded Forth, William Payne C050 - $20
*** IBM, 4.3 Mb

68HC11 Collection C060 - $16
Collection of Forths, Tools and Floating Point routines for the 68HC11 controller.
*** IBM, 2.5 Mb

F83 V2.01, Mike Pery & Henry Laxen C100 - $20
The newest version, ported to a variety of machines. Editor, assembler, decompiler, metacompiler. Source and shadow screens. Manual available separately (items 217 & 235). For other 83 applications.
* IBM, 83, 490 Kb

F-PC V3.56 & TCOM, Tom Zimmer C200 - $30
* IBM, 83, 3.5Mb

F-PC:TEACH V3.5, Lessons 0-7 Jack Brown C201 - $8
Forth classroom on disk. First seven lessons on learning Forth, from Jack Brown of B.C. Institute of Technology.
* IBM, F-PC, 790 Kb

New Version Replacement Policy

Return the old version with the FIG labels and get a new version replacement for 1/2 the current version price.

MISCELLANEOUS

T-SHIRT "May The Forth Be With You"
Specify size: Small, Medium, Large, Extra-Large on orderform
White design on a dark blue shirt.

POSTER (Oct., 1980 BYTE cover)

FORTH-83 HANDY REFERENCE CARD

FORTH-83 STANDARD

BIBLIOGRAPHY OF FORTH REFERENCES

For faster service, fax your orders 510-535-1295
MORE ON FORTH ENGINES

Volume 10 January 1989 $10 - $15 1#
RTX reprints from 1988 Rochester Forth Conference, object-oriented cmForth, lesser forth engines. 87 pgs

Volume 11 July 1989 $11 - $15 1#
RTX supplement to Footsteps in an Empty Valley, SC32, 32-bit Forth engine, RTX interrupts utility. 93 pgs

Volume 12 April 1990 $12 - $15 1#
Shibley Chip architecture and instructions, Neural Computing Module, NGM227, pips/forth, binary radix sort on 80286, 68010, and RTX2000. 87 pgs

Volume 13 October 1990 $13 - $15 1#
PAIs of the RTX2000 Mini-BEE, EBForth, A2Forth, RTX-2101, 8086 eForth, 8051 eForth. 107 pgs

Volume 14 $14 - $15 1#
RTX Pocket-Scope, eForth for mpP20, Shibley, eForth for CP/M & Z80, XMODEM for eForth. 110 pgs

Volume 15 $15 - $15 1#
Moore: New CAD System for Chip Design, A portrait of the P300; Sibley: Q51 Forth Processor, Q52, RISCing in all; P20 eForth Software Simulator/ Debugger. 94 pgs

Volume 16 $16 - $15 1#
OK-CAD System, mpP20, eForth System Words. 386 eForth, 80386 Protected Mode Operation, FRP 1600 - 16Bit Real Time Processor. 104 pgs

DR. DOBB’S JOURNAL

Annual Forth issue, includes code for various Forth applications.

Sept. 1982 422 - $5 1#
Sept. 1983 423 - $5 1#
Sept. 1984 424 - $5 1#

FORTH INTEREST GROUP

P.O. BOX 2154 OAKLAND, CALIFORNIA 94621 510-89-FORTH 510-535-1295 (FAX)

MAIL ORDERS:
Forth Interest Group
P.O. Box 2154
Oakland, CA 94621

PHONE ORDERS:
510-535-FORTH (check card orders, customer service)
HOURS: Mon-Fri. 9-5 p.m.

PAYMENT MUST ACCOMPANY ALL ORDERS

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

POSTAGE: All orders calculate postage at number of as times selected postage rate. Special handling available upon request.

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

FLAT RATE POSTAGE:
*New $41/46/52

*MEMBERSHIP IN THE FORTH INTEREST GROUP

The Forth Interest Group (FIG) is a world-wide, non-profit, member-supported organization with over 1,500 members and 60 chapters. Your membership includes a subscription to the bi-monthly magazine Forth Dimensions. FIG also offers its members an on-line data base, a large selection of Forth literature and other services. Cost is $40 per year for U.S.A. & Canada surface; $46 Canada air mail; all other countries $50 per year. This fee includes 80364248 for Forth Dimensions. No sales tax, handling fee, or discount on membership.

When you join, your first issue will arrive in four to six weeks; subsequent issues will be mailed to you every other month as they are published—six issues in all. Your membership entitles you to a 10% discount on publications and functions of FIG. Dues are not deductible as a charitable contribution for U.S. Federal income tax purposes, but may be deductible as a business expense.

MAIL ORDERS:
Forth Interest Group
P.O. Box 2154
Oakland, CA 94621

PHONE ORDERS:
510-535-FORTH (check card orders, customer service)
HOURS: Mon-Fri. 9-5 p.m.

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

POSTAGE: All orders calculate postage at number of as times selected postage rate. Special handling available upon request.

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

FLAT RATE POSTAGE:
*New $41/46/52

For faster service, fax your orders 510-535-1295

XIV-6
the address into memory. This is accomplished by the phrase:

```
SMUDGE -FIND
IF DROP , ELSE 0 ERROR THEN
SMUDGE
```

(It is possible to dispense with the necessity for naming the namesake word by playing with the contents of the user variable IN to neo- and mezooforthwrights]. The implementation will be left as an exercise for the student. It was not implemented to save space in the dictionary, not because the author was lazy.)

Another general caveat is that the optimizer must not optimize across branch terminations. While it might be acceptable to optimize the phrase FOO F8, the phrase FOO THEN F8 is not readily optimized. As THEN is an immediate word and leaves nothing in the dictionary where the optimizer can detect its passage, we must redefine it to leave a flag. This is done on screen 585. This is why the run-time portions of our optimizers examine the variable OPT immediately after they examine STATE.

Two defining words have been produced. UNARY is used to optimize words which are unary operators. That is, they take one item from the stack and operate on it, leaving one or zero items on the stack. BINARY is for words which take two items on the stack, and leave one. For examples of daughter words, see screen 589.

The Implementation

With the basic concepts laid down, we can expand our optimizer in three ways. We can add new defining words, for new classes of optimizers. We can add new daughter words to the existing defining words. We can add new capabilities and, if needed, new parameters to the existing defining words and their daughter words.

The last method of extension is how the optimizer words were produced in the first place. The programmer started out with a default action (compile the namesake, as usual), and one test and one action for a desired condition. As new phrases were considered for optimization, the nesting of IF ... ELSE ... THEN clauses continued apace.

This methodology allowed for incremental testing of the words under development. Screen 590 shows a test for the binary operator AND. The test is done by compiling two words. One is a code definition, consisting of the desired output for the compiler. The other is a test high-level word which exercises the optimizer. Screens 591 and 592, not shown, contain the target defining word and daughter words.

The last two lines of the screen compare the two words and disassemble \( \text{decompile} \) them both automatically as part of the compilation process. These two tests almost instantly indicate problem areas with words under development. Automated testing of compiler output in this manner allowed very fast, reliable development of the optimizers, and was essential to the success of the project.

Once the basics of the optimizing code have been worked out, it remains only to incrementally add functions to analyze the code and handle the phrases where optimization is desired.

Selecting Phrases for Optimization

If you have your own target compiler and nucleus source, the best way to optimize all possible applications is to improve the nucleus. Anything that improves BLOCK will improve words that call BLOCK. So as FastForth was developed, optimizers were added to the target compiler as well as to the FastForth environment. The choice of phrases to optimize reflects an effort to improve the nucleus first, with improvements elsewhere secondary.

As noted, the phrase \(<\text{USER VARIABLE}>\) F8 shows up all over the nucleus. Similarly, \(<\text{USER VARIABLE}>\) F1, \(<\text{USER VARIABLE}>\) OFF and \(<\text{USER VARIABLE}>\) 1+. The optimizations of F8 and F1 were primary, with the others secondary. These are the phrases to be optimized by the optimizer defining word UNARY, on screens 586 and 587.

These words also operate with variables and often with constants. Both variables and constants compile to in-line literals, either in the form of \(<\text{value}>\) @LS-\{ MOV, or in the form of \(<\text{value}>\) #DR7 MOVQ, DR7 S-\{ MOV, for literals in the range of (hex) \(-80\) to \(7F\). However, since most variables and constants used as variables will be long values, it is essential to detect long literals, with short ones a possible addition for the student.

The long literal form compiles into:

\(<\text{value}>\) @@L-\{ MOV,

After manipulation by F8 the code should look like this:

\(<\text{value}>\) @@L AR0 MOV, AR0 [ S -\{ MOV,

After manipulation by F1 the code should look like this:

\(<\text{value}>\) @@L AR0 MOV, S [ AR0 [ MOV,

This means that the code in UNARY will twiddle the literal's opcode to change its destination, and lay down a new instruction. Since the instruction will vary with the word being compiled, this must be provided as an operand to each optimizer as it is compiled. This instance is handled on screen 587, lines three and four.

With nuclear optimization in mind, the phrase \(<\text{USER VARIABLE}>\) F8 F8 is handled as well. This phrase shows up in places that effect compiler speed, such as in -FIND or LATEST. Any applications which use double indirection will benefit.

The next defining word for optimizers is the family of binary words. These are words which, prior to optimization, take two operands from the stack and return one. These are +, -, etc., as indicated on screen 589. In code they take the form:

\( S[ DR7 MOV, DR7 S[<\text{opcode}>] \), NEXT

If we can detect literals and user variables, and see to it that their contents are left in DR7, we can then compile the appropriate opcode to complete the operation, saving a push
to and a pop from the data stack.

For example, adding a byte literal to the top of the stack becomes:
```
<value> # DR7 MOVQ, DR7 S [ ADD.
```

Similarly, adding the contents of a user variable to the top of the stack goes from:
```
<user variable> U &{ DR0 MOV, DR0 S -{ MOV, S + DR7 MOV, DR7 S [ ADD.
```

to:
```
<user variable> U &{ S [ ADD.
```

This optimization gets rid of three instructions and produces an optimization of fewer instructions than original source words. Not bad for not being an example of Moorish architecture.

To return to the original example, an updated table taking into account the optimizer is as follows:

<table>
<thead>
<tr>
<th>Item</th>
<th>JSR/BSR w/ Optimizer</th>
<th>Indirect Threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>code field</td>
<td>0 bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>SCR</td>
<td>Part of a four-byte instruction laid down in line.</td>
<td>Four bytes</td>
</tr>
<tr>
<td>F@</td>
<td>The rest of the four-byte instruction.</td>
<td>Four bytes</td>
</tr>
<tr>
<td>LIST</td>
<td>Two, four or six bytes of BRA or JMP.</td>
<td>Four bytes</td>
</tr>
<tr>
<td>exit code</td>
<td>0 bytes</td>
<td>Four bytes</td>
</tr>
<tr>
<td>Total</td>
<td>10 bytes, maximum</td>
<td>20 bytes</td>
</tr>
</tbody>
</table>

Comparison: Traditional Compilers

A conceptually simple but very powerful Forth code optimizer can be had in five screens, less than two pages. One has problems imagining a traditional compiler with optimization occupying so small a source code space. Also, one has a hard time imagining the likes of AT&T or Microsoft releasing source for their compilers. And you don't have to call a 900 number to get support.

Furthermore, the optimizer presented here is a complete unit, and can be removed from the FastForth environment without any changes except, of course, in the size and speed of the generated code. It is dependent only upon the nature of the target processor.

Additional phrases may be selected for optimization by the user, who need only add them to the compiler in the traditional Forth manner. Eventually, a diminishing return of better speed and code size must be offset against development time and costs. Unlike the traditional compiler, this tradeoff may be made by the end user, the application programmer, if he wishes. In fine Forth tradition, the application programmer may modify the compiler to suit his application, rather than the usual methodology of modifying the application to fit the compiler's procrustean bed.

Indeed, the very notion of an application programmer having the ability to modify his compiler is a heresy to the ayatollahs of traditional computing.

Conclusions

The FastForth code optimizer produces fast, efficient code. It is easy to understand, and can be modified readily by the end user. It is very powerful and conceptually very simple. Indeed, anyone reasonably familiar with the instruction set of his target processor and the inner workings of his Forth can write one. Like Forth itself, it makes an abattoir of the sacred cows of computing.

Availability

In the best Forth tradition, the code is released to the public domain. Enjoy it in good health.

FastForth for the Atari ST, including the above code, may be had in alpha release from the author, Charles Curley, P.O. Box 2071, Gillette, Wyoming 82717-2071. Please consult the author for the current state of documentation, etc.


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, OS/2, and the 80386
- 16-bit and 32-bit implementations
- Full screen editor and assembler
- Uses standard operating system files
- 500 page manual written in plain English
- Support for graphics, floating point, native code generation

For Applications: Forth-83 Metacomplier
- 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, 8088, 88000, 6502, 8051, 8096, 1802, 6303, 6809, 68HC11, 34010, V25, RTX-2000
- No license fee or royalty for compiled applications

Laboratory Microsystems Incorporated
Post Office Box 10430, Marina del Rey, CA 90295
Phone Credit Card Orders to: (310) 300-7412
FAX: (310) 301-2751
Optimizing Forth

0 ( optimizers for : defs
1 BASE F@ HEX
2 0 VARIABLE OPT ( not particularly re-entrant! )
3
4 : THEN HERE OPT F! [COMPILE] THEN ; IMMEDIATE
5
6 : BEGIN HERE OPT F! [COMPILE] BEGIN ; IMMEDIATE
7
8 : OPGET ( addr ct --- | get operand ct bytes from addr )
9 + W@ ;
10
11
12 -->
13
14
15

Scr # 586

0 ( optimizers: unary
1 : UNARY CREATE SMUDGE -FIND IF DROP , ELSE 0 ERROR THEN
2 SMUDGE W, W, W, IMMEDIATE
3 DOES> STATE F@ ( only if compiling... )
4 IF HERE OPT F@ - ( not following a begin)
5 IF HERE 6 - W@ 273C - ( following a literal? )
6 IF 4 OPGET HERE 6 - W! ( yyy ** @#1 xxx, )
7 ELSE HERE 2- - W@ 2708 = ( ar0 s - [ mov, eg user )
8 IF -2 ALLOT HERE 4- - W@ 41EE = ( user variable? )
9 IF 6 OPGET HERE 4- - W! ( yyy u ** @ [ xxx, )
10 ELSE 8 OPGET W, THEN [ ( yyy ar0 [ xxx, )
11 -->
12
13
14
15

Scr # 587

0 ( optimizers: unary
1 ] ELSE HERE 4- - W@ 272E = ( user f@ optimize )
2 IF 206E HERE 4- - W! 8 OPGET W,
3 ELSE HERE 6 - W@ 2739 = ( literal f@ optimize )
4 IF 2079 HERE 6 - W! 0 OPGET W,
5 ELSE F@ <COMP> THEN THEN THEN THEN
6 ELSE F@ <COMP> THEN ( following br resolution )
7 ELSE @EXECUTE THEN ; ( not compiling )
8
9
10 : BINARY CREATE SMUDGE -FIND IF DROP , ELSE 0 ERROR THEN
11 SMUDGE W, W, IMMEDIATE
12 DOES> STATE F@ ( only if compiling... )
13 -->
14
15

c) 1985-92 by Charles Curley

Tuesday 6/10/92 11:20:08

(For the Dimensions continued on next page.)
Scr # 588

0 ( binary defining word
1 ) IF HERE OPT F@ - ( not following a begin)
2 IF HERE 4- C@ 70 = ( byte literal?)
3 IF HERE 4- E TOGGLE ( xx # dr7 moveq,
4 -2 ALLOT 4 OPGET W, ( dr7 s [ xxx,
5 ELSE HERE 6 - W@ 273C = ( large literal?
6 IF 6 OPGET HERE 6 - W! ( yy #1 s [ xxx,
7 ELSE HERE 4- W@ 272E = ( user f@ ??
8 IF HERE 4- 4 TOGGLE 4 OPGET W, ( ofuser s [ add)
9 ELSE HERE 6 - W@ 2739 = ( literal f@ ??
10 IF HERE 6 - 9 TOGGLE 4 OPGET W, ( lit dr7 mov,
11 ELSE F@ <COMP> THEN THEN THEN THEN
12 ELSE F@ <COMP> THEN ( following br resolution
13 ELSE @EXECUTE THEN ; ( not compiling
14 -->
15

Scr # 589

0 ( daughter words
1 ( opget 8 6 4
2 5290 52AE 52B9 UNARY 1+! 1!
3 4290 42AE 42B9 UNARY OFF OFF
4 209B 2D5B 23DB UNARY F! F!
5 2710 272E 2739 UNARY F@ F@ F@
6
7 ( l.w. lit byte lit
8 693 DF93 BINARY + +
9 493 9F93 BINARY - -
10 93 8F93 BINARY OR OR
11 293 CF93 BINARY AND AND
12 A93 BF93 BINARY XOR XOR
13 BASE F!
14
15

Scr # 590

0 \ test area for macro mods
1 DEBUG FORTH DEFINITIONS FORGET TASK
2 : TASK ; BASE F@ >R HEX
4 0 VARIABLE SNARK
5 CODE FOO ofuser fld dr7 mov, dr7 s [ and,
6 snark @#1 dr7 mov, dr7 s [ and,
7 7f # dr7 moveq, dr7 s [ and,
8 ffff #1 s [ and,
9 NEXT ;C
10
11 1 2 +THRU
12 : BAR f1d f@ and snark f@ and 7f and ffff and ;
13
15 ' FOO DUP 2- W@ ' BAR EDITOR -CITEXT . UN: BAR UN: FOO ;S

fastForth on Atari ST (c) 1985-92 by Charles Curley
Tuesday 6/10/92 11:20:14
Math—Who Needs It?

Prof. Tim Hendtlass
Hawthorn, Australia

The title of this article is a deliberate double entendre. Whatever one's feelings about mathematics in general, arithmetic (at least) is going to be needed sooner or later in your programs. One of the most striking things about Forth, quickly noticed by people who are used to another language, is that 16-bit integers are the only types of numbers apparently directly supported in basic Forth. A closer inspection shows that this is not strictly true, but certainly there are no floating-point numbers defined in the core words of Forth. The reason is, of course, that you can add anything you might want or need to Forth, so why saddle people with things they may not need? If floating point is really required, for example, you just add it, to whatever accuracy you need. The collection of routines in this article are my compilation of math words with varying precision, speed, and portability. I did not write all of them and have gratefully acknowledged the original authors in the text.

Before rushing in to add new math words with extra capabilities, it is wise to see if these capabilities are really needed. In some situations certainly, but not in others. Since, provided the same algorithms are used, floating-point math executes more slowly than fixed-point math, and fixed-point math executes more slowly than double precision, and double precision executes more slowly than single-precision math, it makes sense from the point of view of speed not to use any more capability than you need. Also, the code size in bytes will vary depending on the precision of the math you use, and whether it is written in high-level Forth or mainly in assembly language. As well as the code, there are tables showing the relative speeds and memory requirements of the words described, this is to allow the reader to pick the one that best meets the requirements of the task at hand.

First let us define a couple of terms concerning the representation of numbers: the resolution and the range. The resolution is the minimum possible change that can be represented in a number format. For integers it is one. The range is the difference between the largest and smallest (or, in the case of signed numbers, the most negative) numbers that can be expressed. Integers' resolution is always one, and the range goes up as the number of bits in the integer increases.

For fixed-point numbers, the number is expressed in a single quantity. Depending on how many bits of this quantity you allocate for the fixed decimal places, the resolution and the range vary inversely (e.g., the greater the resolution, the smaller the range). Fixed-point math is very closely related to integer math, except that all numbers are stored internally after having been multiplied by an integer scaling factor. They are divided by this scaling factor before being output. This allows a number of decimal places to be provided while still treating the numbers as integers. Since you still represent numbers in (say) 32 bits, the actual range would be that for 32-bit integers divided by the scaling factor. See Table One for signed numbers, for which the range is the difference between the largest and smallest numbers that can be represented. (For unsigned integers the range would be the same, but from zero to one more than twice the value shown under "largest positive number."

Floating-point numbers are stored in two parts, one expressing an integer number and the other the power of ten (usually) to which this integer should be raised to give the final number. If this power (the exponent) is positive, the number represented can be very large and the resolution small (ten to the power of the exponent). If this power is negative, the number represented can be very small and the resolution high. Using floating-point representation, this tradeoff between range and resolution can alter dynamically without any explicit attention by the programmer as the magnitude of the numbers being used changes.

Single-precision Integer Arithmetic

This is fully provided in F-PC, as in all Forths. The largest positive signed number that can be represented in 16 bits is +32767 and the largest negative signed number is -32768. The smallest number is zero. Of course, since we are dealing with integers, no decimal points are allowed. The four basic functions (add, subtract, multiply, and divide) are provided, plus modulus (MOD), absolute (ABS), and special routines to multiply or divide by two (2* and 2/). In binary, multiplying and dividing by two are the same as just shifting all bits in the number left and right, respectively, by one place. In the case of a left shift, the bit moved into the least significant place is zero; in the case of a right shift, the bit moved in as the most significant bit must be the same as the previous most significant bit in order to preserve the sign of the number.
Multiply two double-precision numbers to give a double-precision product.

Unsigned with overflow check.

: UD*C ( ud1 ud2 -- ud3 )  
  dup>r rot dup>r >r over >r  
  >r swap dup>r  
  um*  
  0 2r> um* d+ 2r> um* d+  
  0 2r> um* d+  
  or 0< abort" D* overflow" ;

Unsigned without overflow check.

: UD* ( ud1 ud2 -- ud3 )  
  rot >r over >r >r over >r  
  um*  
  2r> * 2r> * + + ;

Signed with or without overflow check (replace ud* by ud*c to check for overflow).

: D* ( d1 d2 -- d3 )  
  dup>r dabs 2swap dup>r dabs  
  ud*  
  2r> xor ?dnegate ;

Division (U0 * 2^16 \pm U1) / (V0 * 2^16 + V1) - (A0 * 2^16 + A1)  

\ Use fast algorithm, remainder requires an additional 32-bit multiplication and subtraction.  

: T* ( ut un -- ut )  
  dup rot um* 2>r  
  um* 0 2r> d+  
  ;

: T/ ( ut un -- ud )  
  >r r@ um/mod swap  
  rot 0 r@ um/mod swap  
  rot r> um/mod swap drop  
  0 2swap swap d+  
  ;

: U*/ ( ud un1 un2 -- ud2 )  
  >r t* >r t/  
  ;

: UD/ ( Uh U0 VlV0 -- A1 A0 )  
  dup 0=  
  if swap t/  
  else  
    dup 65536. rot 1+ um/mod >r  
    drop r@ t* drop 2>r  
    dup 0 2r> u*/ d-  
    2r> r> -rot nip u*/  
    nip 0  
    then  
  ;

: D/MOD ( dnl dn2 -- drem dquot )  
  2 pick over xor >r  
  dabs 2swap dabs 2swap  
  4dup ud/ 2dup 2>r  
  ud* d-  
  2r> t> ?dnegate  
  ;

: D/ ( dnl dn2 -- dquot )  
  2 pick over xor >r  
  dabs 2swap dabs 2swap  
  ud/  
  t> ?dnegate  
  ;
Double-precision Integer Arithmetic

A limited double-precision capability is built into all
Forths with double-number extensions, and F-PC is no
exception. A double-precision number is one that is
expressed in 32 bits, rather than the 16 bits of a single-precision
number. Since these are still integers, double-precision
numbers can represent much larger numbers,
from
\[ 2^{14} \times 10^{-16} \text{ to } 2^{14} \times 10^{16} \]\nfor single-precision, and
\[ 2^{14} \times 10^{-32} \text{ to } 2^{14} \times 10^{32} \]\nfor double-precision, in fact. When do you need them?
When you can't express what you want with single
precision, naturally.

For example, suppose you wanted to
store the number of cents you made per year; in all
probability, 16 bits would not be enough, as it would only
allow you to earn up to about 320 per year. If you think
about that example, it may occur to you that, since cents are
the fractional parts of a dollar, you have a sort of two-decimal
place, fixed-point arithmetic here. As long as you add or
subtract numbers, the fixed implied decimal point will stay
in place; but if you multiply or divide, the implied decimal
point gets messed up. Below we will see how to correct that,
but first let us consider what double-precision integer
facilities are provided.

Of the four basic functions, only addition (D+) and
subtraction (D-) are provided directly; in a moment we will
generate D* and D/ (among others). To print a double
number, there is D, (and its formatted cousin D.R). A
double-precision absolute value word is provided (DABS).
There are also limited double-precision comparisons: D=, D>,
D<, and DDE=. To input a double number, either from the
keyboard or in line in a definition, all you need to do is put
a decimal point in the number somewhere. This use of a
decimal point to indicate a double number can lead to
misunderstanding. It is intended for when you are using an
implied fixed decimal place, but it often misleads people into
believing that the decimal part will be correctly handled. It
won't, unless you specifically use words that do so. If you
were to enter the number 31415., the number in the two
positions on the stack would be no different than if you had
entered 3.1415. However, the number of digits after the
decimal place is recorded in the system variable DPL,
especially for when you need this information. (As the same
variable is used for all number input, you had better collect
the value from DPL and use it, or put it somewhere safe
before the next number arrives.) In the first case above, DPL
would contain zero; in the second case four.

The main words we need to add to flesh out our double-
precision integer capability are D* and D/. D* may produce
an answer that is too big to fit into 32 bits (just as * may
produce an answer too big to fit in 16 bits). It is possible to
provide a run-time check to detect this (just make sure that
the top 32 bits of the answer are zero), but this takes time.
If you are sure that overflow will not occur in a problem, there
is no need to calculate the top 32 bits of the answer. Code
to perform 32-bit by 32-bit multiplication, with and without
overflow check, is given below. In each case, we do
unsigned arithmetic (both numbers are assumed positive);
for signed arithmetic, we work out the sign of the answer,
make both numbers positive, do the multiplication, and then
apply the correct answer sign.

The algorithm for 32-bit multiplication is built from the 16-
bit multiplication we already know how to do. Consider the
following,

\[ (a \times 2^{16} + b) \times (c \times 2^{16} + d) = (a \times c) \times 2^{32} + (b \times c + a \times d) \times 2^{16} + b \times d \]

\( (a \times 2^{16} + b) \) is one 32-bit number and \( (c \times 2^{16} + d) \) is the other.
Note by expanding it we have reduced one 32-bit by 32-bit
multiply to four 16-bit by 16-bit multiplies, which we know
how to do.

If we want to perform an overflow check, we get the full
32-bit answer by doing four 16-bit multiplies, offsetting their
answers by the correct number of bits, and adding. The result
is a 64-bit (i.e., quad-precision) number. If the numbers were
both positive and the top 32 bits of the result are not zero,
the result was too big to fit into 32 bits.

If an overflow check is not needed, we proceed by noting
that a*c must equal zero (otherwise the result would not fit

Table One.

<table>
<thead>
<tr>
<th>Word</th>
<th>Decimal places</th>
<th>Scaling Factor</th>
<th>Largest positive number</th>
<th>Largest negative number</th>
<th>Smallest increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>16</td>
<td>0</td>
<td>na</td>
<td>32767</td>
<td>-32768</td>
</tr>
<tr>
<td>Integer</td>
<td>32</td>
<td>0</td>
<td>na</td>
<td>2,147,483,647</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>Fixed point</td>
<td>32</td>
<td>1</td>
<td>10</td>
<td>214,748,364.7</td>
<td>-214,748,364.8</td>
</tr>
<tr>
<td>Fixed point</td>
<td>32</td>
<td>2</td>
<td>100</td>
<td>21,474,836.47</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>Fixed point</td>
<td>32</td>
<td>3</td>
<td>1000</td>
<td>2,147,483,647</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>Fixed point</td>
<td>32</td>
<td>4</td>
<td>10000</td>
<td>214,748,364.7</td>
<td>-214,748,364.8</td>
</tr>
</tbody>
</table>
Defining the fixed-point structure

```
VARIABLE FDPL
VARIABLE FSCL
: FPLACES ( -- n ) fdpl @ ;
: FSCL ( -- n ) fscl @ ;
: FIXED ( n -- )
  0 max 4 min fdpl !
  1 fdpl @ 0 ?do 10 * loop fscl !
  
; 3 FIXED

Outputting numbers
: (F.) ( fn -- adr len )
  tuck dabs <# hold
  fdpl @ 0 ?do # loop
  ascii . hold
  # hold
  rot sign #>
  ;
: F. ( fn -- ) (f.) type ;
: F.R ( fn p -- )
  >r (f.) r> over - 0 ?do bl
```

Inputting numbers
```
: D10* ( d1 -- 10^d1 )
  d2* 2dup d2* d2* d+
  ;
: FIX ( dn -- fn )
  dpl @ 0<
  if s>d 0 dpl ! then
  dpl @ fplaces <-
  if dpl @ fplaces <
    if fplaces dpl @ ?do d10* loop
    else abort" Too many decimal places"
  then
  then
```

Multiply two fixed-point numbers, producing a fixed-point result.
```
: FIX* ( f1 f2 -- f1*f2 )
  rot 2dup xor >r
  -rot dabs 2swap dabs
  dup>r rot dup>r r> over -r
  >r swap dup>r
  um*
  0 2r> um* d+ 2r> um* d+
  2r> * +
  fscale mu/mod
  0< abort" Fixed * Overflow!" >r
  fscale mu/mod rot drop
  r> + r> ?dnegate
```

Divide two fixed-point numbers, producing a fixed-point result.
```
: FIX/ ( f1 f2 -- fquot=f1/f2 )
  2 pick over xor >r
  dabs 2swap dabs 2swap
  2dup >r >r
  d/mod fscale 0 d*
  2swap fscale 0 d*
  r> r> d/
  d+
  r> ?dnegate
```

\ holds number of implied decimal places
\ holds the scaling factor we are using
\ return number of implied decimal places
\ return the scaling factor we are using
\ clip to between 0 and 4 decimal places
\ store #places, calc. & store scaling factor
\ default to three decimal places
\ prepare fixed-point # ready to output
\ keep copy of top byte so we know sign
\ convert to positive number
\ start conversion with a leading blank
\ convert places after decimal point
\ put a decimal point in place
\ convert integer part
\ put sign in place, tidy stack
\ print fixed-point number
\ print right justified in a field of p places
\ convert, pad with blanks as needed, then type
\ multiply a 32-bit number by 10
\ 8*d+2*d=10*d
\ single or double number?
\ if single, convert to double
\ # decimal places entered not fplaces?
\ too few places specified?
\ yes, too few so scale the number up
\ no, too many - we can't handle this
\ sign of answer to return stack
\ make both numbers positive
\ put a c c b on return stack
\ put a d onto return stack
\ b*d
\ offset 16 bits, add on a*d+b*c
\ add on low byte of a*c
\ divide ms32 bits, ans to R.
\ unless overflow quotient to R....
\ divide remainder and last 16 bits
\ assemble final answer, negate if required
\ Divide two fixed-point numbers
\ work out sign of answer and save
\ make all numbers positive
\ keep copy of divisor
\ scale integer part of answer
\ and then scale remainder
\ divide remainder by divisor
\ add frac part of ans
\ put on final sign
into 32 bits, so there is no point in performing this multiply. Similarly \((bc+ad)\) must give an answer that is no bigger than 16 bits. So only \(b+d\) need be done to 32-bit precision, and \((bc+ad)\) to 16-bit precision, and \(a\) need not be done at all. Naturally, this makes this version faster than the one with overflow check.

The traditional method to perform a 32-bit by 32-bit division is by a subtract-and-shift algorithm (the way we were taught at school, except bit by bit rather than digit by digit), which gives both the result and the remainder. This method can be used to provide division of any precision, not just 32 bits. The method shown here uses an algorithm designed (only) for 31-bit unsigned numbers (that is, 32-bit signed numbers without the sign). The advantage of this new algorithm is speed: it is more than twice as fast. The algorithm is described in Knuth's book, but I came across it first in an article by Nathaniel Grossman in *Forth Dimensions*. I have recoded it completely for faster execution.

The algorithm works as follows. Let the dividend be \(U_0*2^{16} + U_1\) and the divisor be \(V_0*2^{16} + V_1\). Also let \(D\) be a large integer not bigger than 65536/\(V_0\). For simplicity of calculation, let \(D = 65536/(V_0-1)\) as suggested by Knuth. Then our division sum is:

\[
\frac{U_0*2^{16} + U_1}{V_0*2^{16} + V_1} = \frac{D \times (U_0*2^{16} + U_1)}{W_0*2^{16} + W_1}
\]

where \(D \times (V_0*2^{16} + V_1) = W_0*2^{16} + W_1\)

and

\[
\frac{U_0*2^{16} + U_1}{V_0*2^{16} + V_1} = \frac{D \times (U_0*2^{16} + U_1) - U_0 \times W_1}{W_0}
\]

plus an error term.

The error term is so small it may be ignored, unless we wished to calculate the remainder. In practice, it is simpler to find the remainder (if we need it) by taking away the product of the answer and the divisor from the dividend. Also, we must check that \(V_0\) is not zero; if it is, we must not use the relationship above, as we will be trying to divide by zero. However, if \(V_0\) is zero, our problem is reduced to dividing a 32-bit number by a 16-bit number, a much simpler task.

The code in Figure One implements the various versions of \(D\times D\) in a straightforward way.

32-bit Fixed-point Arithmetic

The software to be described will allow you to choose the number of decimal places you want and, therefore, the scaling factor that will be used. The more decimal places you want, the smaller the largest positive and negative numbers you can handle, but the smaller the smallest number increment you can represent.

To perform fixed-point math, only the number input, number output, multiplication, and division words need to be changed. The addition, subtraction, and absolute value double-precision words still work. First you must decide how many decimal places you want to the right of the decimal point; for simplicity, let us call this \(N\). Any number that does not have this number of decimal digits must be multiplied by the appropriate power of ten to get its implied decimal point to line up with all the others. After a normal double-precision multiply, the 64-bit answer will be too large by \(10^N\), so to get the correct answer simply requires a division by \(10^N\).

Dividing by 10 is not as easy as dividing by two, unfortunately, so this extra step adds a bit to the execution time.

After a division, the result will be too small by \(10^N\). But just doing the division and then multiplying by \(10^N\) would lose precision. We must do the division, scale the remainder up by \(10^N\), do an integer division of this remainder, and add this result to the previous result to get a final result to the fullest precision possible.

The word to print a fixed-point number, \(P\), (or \(F.R\) to print the number right justified in a specified field), really prints two numbers: a number representing the integer part and a second representing the fractional part. These are printed with a decimal point in between (and leading blanks, as required, in the case of \(F.R\)).

In this simple package, the user has to specify with the word FTX that the number just entered is to be a fixed decimal point number. From the keyboard, this would be done by entering 123.4 FTX, for example. To put the same fixed-point number in a colon definition, you would specify it as \([123.4\ FIX]\ DLITERAL.

The code in Figure Two implements these words in a straightforward way. By default, the number of implied decimal places is set to three; modify the line 3 FIXED to alter the number of implied decimal places to any integer between zero and four.

32-bit Floating-point Arithmetic

If you need a greater dynamic range of numbers than can be readily accommodated in either 32-bit integer or 32-bit fixed-point arithmetic, but can tolerate lesser basic resolution than 32-bit integers provide, you might consider 32-bit floating point. Here, some of the 32 bits are used to hold an exponent, and the remainder are for the basic number. The code shown below allocates 16 bits each to the basic signed number and the signed exponent. The dynamic range is probably unreasonably high, and one might be tempted to increase the number of bits allocated to the basic number and decrease the number allocated to the exponent. The programming ease of staying with 16-bit quantities for each, and the speed penalty that would be incurred by dealing with smaller parts of the number, strongly dictate otherwise. The accuracy is a little better than four significant digits, about the accuracy of the traditional logarithm tables that school children suffered before the advent of calculators. The code shown below, which implements such a 32-bit floating-point number package, was originally written by Martin Tracy and has only been slightly modified for greater speed by this author. Martin called it "Zen" math. There is also an add-on


Figure Three. 32-bit floating-point math.

\ Trim a double-number mantissa and an exponent of ten to a floating number. 
: TRIM ( dn n = f) 
  \ exponent to return stack 
  \ save copy of sign, make double positive 
  \ MSB low word set or top 16 bits no zero? 
  \ if so, too big to fit into 16 bits when signed
  >r 
  tuck dabs 
  begin over 0< over 0< or 
  while 
  0 10 um/mod >r 10 um/mod nip r> 
  repeat rot ?dnegate drop r> 
  \ and increase exponent 
  \ apply sign and final exponent

: F+ 
  rot 2dup - dup 0< 
  if 
  negate rot >r nip >r swap r> 
  else 
  swap >r nip 
  then 
  >r s>d r> dup 0 
  ?do >r d10* r> 1- 
  over abs 6553 > 
  if leave then 
  loop 
  r> over + >r 
  if rot drop 
  else rot s>d d+ 
  then r> trim 

: F- 
  negate r> ; 

: FNEGATE 
  >r negate r> ; 

\ 32 bit floating-point addition and subtraction 
: F+ ( f1 f2 -- f3 ) 
  \ work out difference in exponents 
  \ top number has the larger exponent 
  \ keep larger and diff, swap mantissas 
  \ top has a smaller or equal exponent 
  \ keep larger (on return stack) and diff 
  \ convert larger to double, top 16 bits >r 
  \ multiply mantissa by 10, decrement exponent 
  \ would a *10 cause overflow of these 16 bits? 
  \ prematurely terminate loop if so
  rot + >r 
  2dup xor >r 
  abs swap abs um* 
  r> ?dnegate r> trim 

: FNEGATE 
  negate f+ ; 

\ 32bit floating point multiplication 
: F* ( f1 f2 -- f3 ) 
  \ calc exp of answer, save on return stack 
  \ save xor of mantissas too (sign of answer) 
  \ make mantissas positive and multiply 
  \ apply sign and then get exponent and trim 
  rot + >r 
  2dup xor -rot 
  abs dup 6553 min rot abs 0 
  begin 2dup d10* nip 3 pick < 
  while d10* r> 1 ->r 
  repeat 2swap drop um/mod 
  nip 0 rot ?dnegate r> trim 
  r> ?dnegate >r

\ 32-bit floating-point division 
: F/ 
  \ check for divide by zero 
  \ get exponent of answer, put on return stack 
  \ get sign of answer, tuck down on stack 
  \ make number +ve, ensure divisor < 6553 
  \ would divisor * 10 be less than dividend? 
  \ yes, divisor * 10, decrement answer exponent 
  \ now do the division 
  \ lose remainder, apply sign get exp and trim 
  over 0= abort" d/0 error!" 
  rot swap - >r 
  2dup xor -rot 
  abs dup 6553 min rot abs 0 
  begin 2dup d10* nip 3 pick < 
  while d10* r> 1 ->r 
  repeat 2swap drop um/mod 
  nip 0 rot ?dnegate r> trim 
  r> ?dnegate >r

\ 32-bit floating-point input and output 
\ Numbers to be floated must include a decimal point when entered. 
\ DPL contains the number of digits entered after the decimal point. 
: FLOAT ( n -- f) 
  \ float the last entered number. 
  dpl @ negate trim 

: F. ( f --) 
  \ print a floating number in fixed format. 
  >r dup abs 0 
  #r ?@ 0 max 0 ?do ascii 0 hold loop 
  ?@ 0< 
  if ?@ negate 0 max 0 ?do # loop ascii . hold 
  then ?drop #s rot sign 
  #> type space ;
to Zen which extends it to calculate transcendental functions (with an accuracy of only about three figures) written by Nathaniel Grossman. This is not reproduced here; it can be found in Dr. Dobbs Toolbook of Forth Volume Two, in the file of these words on GEnie's Forth Round Table, or directly from this author. The code in Figure Three implements Zen math.

### Forth or Assembly Code?

All the words above are written in Forth and are thus able to be transported from machine to machine. There are two reasons why words written in assembly code will run faster. (They will, of course, not be able to be ported to other processors nearly as readily.) One reason is that, although there is only a slight speed overhead involved in using the Forth inner interpreter, this can accumulate to a small but significant sum over enough operations. The second reason is not as obvious, but accounts for more of the speed penalty observed. Forth has no carry; if you add two 16-bit quantities and the sum is too large to fit into 16 bits, the uppermost (17th) bit of the answer is lost. In arithmetic involving more than 16 bits, a carry is needed in order to do the calculations—you have to synthesize one, which takes time. By writing in machine code, you can make direct use of the carry flag of the processor. The 48-bit floating-point package described below is written mainly in assembly language, and is significantly faster than any of the other packages given. Not all of this speed increase comes from using assembler—the algorithms used are highly optimized. If you want the fastest speed arithmetic possible for a given processor, you must use the most efficient algorithms and assembly language. The result will be larger than the simple algorithms described above, and totally non-portable. Of course, a hardware math processor will always perform faster than any software solution on the main processor.

### 48-bit Floating-point Arithmetic, SFLOAT

This is a full software assembly language floating-point package for F-PC written (and copyrighted) by Robert L. Smith. It is in the file SMITH.ZIP which comes as part of the F-PC package. The size of a floating-point number is 48 bits (six bytes). The largest difference to get used to when you load this software is the fact that you now have another (third) stack. Holding the floating-point numbers on the regular data stack would make stack operations an absolute nightmare, so they are given a stack of their own. By default, the floating-point stack is 100 floating-point numbers deep, but you can change this just by altering one constant before you load the software. Words expect their floating-point parameters on the floating-point stack and leave their floating-point results there. Any flags that result from operations on floating-point numbers are left on the normal data stack, and any integers needed are obtained from the normal data stack. Words are provided to manipulate the floating-point stack; the name used is almost always the name of the same operation of the data stack, but with a leading F. Thus, we have FDUP and FROT, for example.

SFLOAT not only provides a full set of arithmetic and transcendental functions, it may also alter the outer interpreter of F-PC. The new outer interpreter allows you to enter floating-point numbers in line. Any number with an embedded decimal point or with an exponent will be converted to a floating-point number. Any number without a decimal point will be treated as a single-precision integer and placed on the data stack. You can control whether you wish to use the normal or the new outer interpreter at any time, by using the words FLOATING and NOFLOATING. A list of words provided by SFLOAT can be found by inspecting the help file that comes with SFLOAT.

### Relative Performance

Shown in Table Two are the timings for addition, subtraction, multiplication, and division for each of the 16- and 32-bit math capabilities shown above. All times are relative, with a 16-bit signed add used as reference, and have been rounded to two significant figures. The times were calculated by timing a loop that performed the required operation 65,536 times, and deducting the time for an empty loop. The actual times you get will depend on the processor speed; on my trusty old 25 MHz '386SX, a 16-bit signed add took about six microseconds. Also shown are timings for SFLOAT. Just looking at the figures can be misleading, as you may be unintentionally equating apples with oranges, so a number of explanatory comments are given below.

The multiply and divide times in row one are small, as the PC processor has hardware 16-bit integer multiply and divide. The far larger times for multiplication and division in row two show the penalty to be paid when you have to

**Table Two.**

<table>
<thead>
<tr>
<th>Description</th>
<th>Add</th>
<th>Subtract</th>
<th>Multiply</th>
<th>Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit signed integer,</td>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>1.3</td>
</tr>
<tr>
<td>written in Forth, portable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit unsigned integer,</td>
<td>2.4</td>
<td>3.8</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>written in Forth, portable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit signed integer,</td>
<td>2.4</td>
<td>3.8</td>
<td>13.1</td>
<td>19</td>
</tr>
<tr>
<td>written in Forth, portable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit fixed point,</td>
<td>2.4</td>
<td>3.8</td>
<td>36</td>
<td>93</td>
</tr>
<tr>
<td>3 decimal places,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>written in Forth, portable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Zen floating point,</td>
<td>19</td>
<td>22</td>
<td>16</td>
<td>69</td>
</tr>
<tr>
<td>written in Forth, portable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48-bit floating point, SFLOAT,</td>
<td>2.9</td>
<td>3.1</td>
<td>2.1</td>
<td>2.9</td>
</tr>
<tr>
<td>written in assembler, non-portable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Forth Dimensions 33 January 1993 February
synthesize operations on long numbers out of repeated use of short-length operators. Doubling the word size increased the execution time by a much higher factor. Row three shows that just adding the extra code to keep track of the implied decimal point for fixed-point multiplication and division has added about another 50% to the time, except for addition and subtraction, fixed-point arithmetic costs significant time over integer arithmetic.

For curiosity, the multiplication word in row three was rewritten as in-line code. This saves the time used by the inner interpreter (NEXT) and allows intermediate results to be kept in registers instead of being pushed at the end of one word and immediately reloaded again at the start of the next. This new version was faster, but only by about six percent. This modest speed increase must be weighed against the benefits of writing in Forth so that the word is portable to other Forth systems, no matter what the processor. Also, Forth code is much easier to understand and, therefore, to write and debug.

The 32-bit floating-point Zen package results may seem strange. The clue to understanding them lies in the way that a separate exponent simplifies multiplication and division, but complicates addition and subtraction. Since the actual number in Zen is a 16-bit quantity, multiplication is done by multiplying the 16-bit numbers and adding their exponents.

<table>
<thead>
<tr>
<th>Math Package</th>
<th>Memory requirements in bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>header space</td>
</tr>
<tr>
<td>32-bit integer, 4 functions</td>
<td>80</td>
</tr>
<tr>
<td>32-bit fixed point, 4 functions</td>
<td>213</td>
</tr>
<tr>
<td>32-bit floating point, 4 functions</td>
<td>106</td>
</tr>
<tr>
<td>SFLOAT, 4 functions only</td>
<td>671</td>
</tr>
<tr>
<td>SFLOAT full package</td>
<td>2380</td>
</tr>
</tbody>
</table>

For division, the multiplication is replaced by division and the addition by subtraction. As a result, these words are faster than their fixed-point equivalents, which require a 32-bit multiplication and division of the result by a scaling factor. However, addition and subtraction of fixed-point numbers is trivial, while to do the same with floating-point numbers requires that the numbers be shifted (scaled) so that their exponents are equal before the required operation can be done.

The times shown in row six seem little short of amazing, considering that this is for 48-bit floating point, and show what can be done if you abandon the requirement for portability and write in highly optimized machine code. Note again the (relative) inefficiency of addition and subtraction compared to multiplication and division. The routines used are anything but trivial to understand (see the file SFLOAT.TXT, for example, for an explanation of the divide algorithm used). An assembly language routine using the same algorithm for fixed point would be faster than even these floating-point times.

Speed is only one criterion, another is the memory these routines take up. Table Three shows the memory needed in F-PC by each of the math packs. The smaller space quoted for SFLOAT is with only the basic four mathematical functions loaded; the larger figure is for the full package, which includes many more functions. If you have a math coprocessor, there is an equivalent package to SFLOAT called FFLOAT, which also comes with F-PC and which is even faster and smaller. FFLOAT is, of course, totally non-portable.

Choose your math routines after considering your need for speed, precision, size, and portability. No one of them is always the best.


FORTH and Classic Computer Support

For that second view on FORTH applications, check out The Computer Journal. If you run a classic computer (pre-PC-clone) and are interested in finding support, then look no further than TCJ. We have hardware and software projects, plus support for Kaypro, S100, CP/M, 6809's, and embedded controllers.

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 also feature Kaypro items from Micro Cowpooch. 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
Math—Who needs it?

Extra words needed to implement 32-bit integer, fixed, and floating-point arithmetic.
anew 32math

32-bit Integer Arithmetic

Untested: 32-bit Integer Multiplication

32-bit Integer Division

comment:

Forth Dimensions 35 January 1993 February
Calculate \( ud \ast \text{unl} / \text{un2} \). Triple intermediate product.

\[
\text{U*/ ( ud unl un2 -- ud2 )}
\]
\[
> \text{r t* r} > \text{t/ ;}
\]

Unsigned 32-bit by 32-bit divide. No remainder.

\[
\text{UD/ ( ud1 ud2 -- ud3 )}
\]
\[
\text{dup 0= \text{top 16 bits of divisor = 0?}}
\]
\[
\text{if swap t/ \text{make it a triple, do the division}}
\]
\[
\text{else}
\]
\[
\text{dup 65536. \text{rot 1+ um/mod >r \text{work out scaling factor, copy to return stack}}}
\]
\[
\text{drop r* t* drop 2>r \text{scale denominator, move to return stack}}
\]
\[
\text{dup 0 2r0 u*/ d- \text{calculate (U-U0*W1/W0)}}
\]
\[
2r> r> -\text{rot nip u*/ \text{multiply by (D/W0)}}
\]
\[
nip 0 \text{/2^16 (use top 16 bits only), make ans double then}
\]

Divides two double numbers. All numbers are signed doubles.

\[
\text{D/MOD ( dnl dn2 -- drem dquot )}
\]
\[
\text{2 pick xor \text{over >r}}
\]
\[
\text{dabs 2swap dabs 2swap}
\]
\[
\text{4dup ud/ 2dup 2>r}
\]
\[
\text{ud* d-}
\]
\[
2r> r> ?\text{dnegate}
\]

\[
\text{D/ ( dnl dn2 -- dquot )}
\]
\[
\text{2 pick xor >r}
\]
\[
\text{dabs 2swap dabs 2swap}
\]
\[
\text{ud/}
\]
\[
\text{r> ?\text{dnegate}}
\]

---

### 32-bit Fixed-Point Arithmetic

* Defining the fixed-point structure *

**variable fdpl**

\[
\text{FPLACES ( -- n ) fdpl @ ;} \quad \text{number of implied decimal places}
\]
\[
\text{FScale ( -- n ) fscl @ ;} \quad \text{scaling factor we are using}
\]
\[
\text{FIXED ( n -- )} \quad \text{clip to between 0 and 4 decimal places}
\]
\[
\text{0 max 4 min fdpl !} \quad \text{store scaling factor}
\]

3 FIXED

**variable fscl**

**Outputting numbers**

\[
\text{(F.) ( fn -- adr len )}
\]
\[
\text{tuck}
\]
\[
\text{dabs}
\]
\[
\text{<# bi hold}
\]
\[
\text{fplaces 0 ?do # loop}
\]
\[
\text{ascii . hold}
\]
\[
\text{#s}
\]
\[
\text{rot sign #>}
\]

\[
\text{FIX. ( fn -- ) (f.) type ;} \quad \text{print fixed-point number}
\]
: FIX.R ( fn p -- ) \ right justify in a field of p places
 >r (f.) \ convert number
 r> over - 0 ?do bl emit loop type \ pad with blanks as needed

; \ ************************************************************
| \ * Inputting numbers
| \ ************************************************************

: D10* ( d1 -- 10*d1) \ multiply a 32-bit number by 10
 d2* 2dup d2* d2* d+ \ 8*d+2*d=10*d

; \ Convert number to fixed-point number - no check made for numbers too large
 \ Example 1234.5 FIX. To compile a fixed-point number in a : definition, use
 \ [ 1234.5 FIX ] DLITERAL

: FIX ( dn -- fn )
 dpl @ 0< \ single or double number?
 if s>d 0 dpl ! then \ if single convert to double
 dpl @ fplaces <
 if dpl @ fplaces <
 if fplaces dpl @ ?do10* loop \ yes, too few so scale the number up
 else abort" Too many decimal places" \ no, too many - we can't handle this
 then

; \ ************************************************************
| \ * 32-bit Fixed-Point Multiply
| \ ************************************************************

\ Multiply two fixed-point numbers, producing a fixed-point result.

: FIX* ( f1 f2 -- f1*f2 )
 rot 2dup xor >r \ sign of answer to return stack
 -rot dabs 2swap dabs \ make both numbers positive
 dup>r rot dup>r >r over >r \ put a c c b on return stack
 >r swap dup>r \ put a d onto return stack
 um* \ b*d
 0 2r> um* d+ 2r> um* d+
 2r> * + \ offset 16 bits, add on a*d+b*c
 2r> fscale mu/mod >r >r \ divide ms32 bits, ans to R. Remainder on stack
 /fscale mu/mod rot drop \ and that remainder and last 16 bits
 0 r> r> d+ r> + \ assemble final answer
 r> ?dlnegate \ yes, answer valid, negate if required
 else abort" F* Overflow" report if an overflow
 then

; \ ************************************************************
| \ * 32-bit Fixed-Point Divide
| \ ************************************************************

\ Divide two numbers

: FIX/ ( f1 f2 -- fquot=f1/f2 )
 2 pick over xor >r \ work out sign of answer and save
 dabs 2swap dabs 2swap \ make all numbers positive
 2dup >r >r \ keep copy of divisor
 d/mod fscale 0 d* \ scale integer part of answer
 2swap fscale 0 d* \ and then scale remainder
 r> r> d/ \ divide remainder by divisor
 d+ \ add fract part of ans
 r> ?dlnegate \ put on final sign


Forth Dimensions January 1993 February
32-bit Floating-Point Arithmetic
Based on Zen Math by Martin Tracy

Trim a double-number mantissa and an exponent of ten to a floating number.

: TRIM ( dn n = f)
  >r \ exponent to return stack
  tuck dabs \ save copy of high word for sign, make double positive
  begin over 0< over 0<> or \ MSB low word set or top 16 bits no zero?
  if so, too big to fit into 16 bits when signed
  0 10 um/mod >r 10 um/mod nip r> \ divide 32-bit mantissa by 10
  r> 1+ >r \ and increase exponent
  repeat rot ?d negate drop r> \ apply sign and final exponent

: F+ ( f1 f2 -- f3 )
  rot + >r \ work out difference in exponents
  if negate rot >r nip >r swap r> \ keep larger (on return stack) and diff, swap mantissas
  else swap >r nip \ keep larger (on return stack) and diff
  then
  >r s>d r> dup 0 \ convert mantissa to be shift to double
  ?do >r d10* r> 1-
  over abs 6553 > \ would a *10 cause overflow of these 16 bits?
  if leave then
  loop
  r> over + >r \ calculate final exponent
  if rot drop \ 
  else rot s>d d+
  then r> trim \ get final exponent and trim

: FNEGATE ( -- r ) \ add negative of the top value
  >r negate r> ;

: F- ( -- r )
  fnegate f+ \ add negative of the top value

: F* ( f1 f2 -- f3 )
  rot + >r \ calc exp of answer, save on return stack
  2dup xor >r \ save xor of mantissas, too (sign of answer)
  abs swap abs um* \ make mantissas positive and multiply
  r> ?d negate r> trim \ apply sign and then get exponent and trim

: F/ ( -- )
  over 0= abort" d/0 error!" \ check for divide by zero
  rot swap - >r \ get exponent of answer, put on return stack
  2dup xor -rot \ get sign of answer, tuck down on stack
  abs dup 6553 min rot abs 0 \ 
  begin 2dup d10* nip 3 pick < \ now do the division
  while d10* r> 1- >r \ lose remainder, apply sign get exp and trim
  repeat 2swap drop um/mod
  nip 0 rot ?d negate r> trim

Code concludes in next issue with 32-bit floating-point I/O and transcendental functions.
It may be downloaded in its entirety from the Forth software library on GETfile.

January 1993 February
A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

Mike Elola
San Jose, California

From the last volume of Forth Dimensions, I have collected comments that reinforce one another and that speak to Forth and its future. The comments brought to you here have previously appeared in FD’s “Letters to the Editor” or “Best of GEnie” columns.

Not so long ago, I viewed the Forth community as a very divided community that was becoming even more divided. However, the views offered here reveal commonly held values and beliefs. Perhaps these values can also shape our vision about how to promote Forth.

John Wavrik is a professor at the University of California (San Diego, California) who has spoken of the strengths of Forth: “Conventional languages allow data structures only to be created by a limited set of mechanisms built into the language—and then impose further limitations on the status of these structures (how they can be passed to functions, how operators may act on them, etc.).”

He described the Forth advantage as “the ability to accomplish difficult things without fighting the language.” He credits Forth with being the only language that always lets him do whatever he determines must be done, and speaks of fighting the rigid features of other languages (Best of GEnie, FIY XIII/3).

Our concerns are focusing on management issues and the development environment...

FD XIII/5). A theme that others will repeat is the relationship between power and knowledge: “Power in Forth comes, in great measure, from the user’s ability to understand how the system works—and being able to harness that understanding.”

Steve Noll gave his testimonial about Forth’s empowerment of the programmer. Crediting Forth for his speed of development, he briefly described five sophisticated machine-control applications that he completed in four years (Letters, FD XIII/5). Although he had come to Forth “kicking and screaming,” he said he was won over. Given his experience, his support for promoting Forth is a natural one: He suggested that a way to attract others to Forth is for FIG to distribute, market, and provide support for a low-cost Forth.

A winning submission in the programming contest held by FIG U.K. a couple of years back was a tiny editor from Mike Lake. He shared the story of the success of M.A.S.S., a company that converted to Forth around 1985 after BASIC, Pascal, and assembler had all been tried. He mentions that the company has distributed over 12,000 Forth applications worldwide (presumably, in a six-year period). Besides sharing his code with us, Lake described his company’s deepening commitment to Forth, culminating in their development of an in-house Forth that gave them “absolute control” (Letters, FD XIII/3).

Dean Sanderson is a key software engineer with Forth Inc. He had this to say about Forth’s future: “For Forth to survive as a respected language, it must prove its adaptability and change enough to support the concerns of management. These include: Integration, Maintenance, Documentation, Declining cost, Quality Assurance, Configuration, and Scheduling. Though we’ve started late, we can survive by capitalizing on what others have learned” (Best of GEnie, FD XIII/3).

John Edgecombe described Forth as a language that enterprises resort to when conventional methods fail. He sympathizes with companies reluctant to use Forth because of the difficulty of getting good Forth help when they need it. He described why he uses Forth: “...I want something I can understand, that I will maintain, and which is economical of my limited resources” (Letters, FD XIII/1).

Tight, clever code is no longer as commercially valued as it once was. While asserting the prominence of the development environment, Laughing Water discounted the importance of Forth’s compactness in today’s marketplace: “If Forth’s virtues as a general programming language—compactness, speed, interactivity, flexibility (anarchy)—have become old fashioned indeed, and we are frequently superseded by mainstream languages in more fully evolved development environments...” (Letters, FD XIII/1).

By reporting that Macintosh Pascal has earned greater mindshare than Forth because of the environment it offers, Conrad Weyns added his voice to those proclaiming the prominence of the programming environment. This viewpoint asserts that a language such as Borland Pascal is popular due to the tools into which it is embedded rather than due to Pascal.

Weyns also joined those equating power and understanding: “A lot of Forth’s power lies precisely in its accessibility; the ability to extend the compiler and interpreter, to add to it, to use or abuse it...” (Letters, FD XIII/3).

Mitch Bradley of Sun Microsystems said, “C is a viable,
Most of us read several journals each month besides value for us, we must be able to “access” exactly how it works. We feel penalized whenever program code or a survey so that every Forth dialect could be briefly outlined.

Divergence considered a flaw? Some would say that flexibility is the point of using Forth, because Forth offers the freedom to solve problems in novel ways. However, for pragmatic goals such as code reuse and code portability, divergence can indeed be our enemy. We have to be shrewed enough to know when a departure from standard technique will ultimately turn out to be a hindrance to our collective Forth future instead of a competitive advantage that will endure.

Brad Rodriguez shared his struggle to understand metacompling (Letters, FD XIII/3). The understanding he sought finally arrived after he attended an advanced poly-FORTH class. After presenting his struggles at the local FIG chapter meeting, he reports that others were able to unravel the secrets of the technique too.

Such an experience underscores our need for various forms of support. Opportunities to receive structured training are helpful, along with informal meetings. Rodriguez’ experience also says something about our values and our requirements as programmers: Before something truly has value for us, we must be able to “access” exactly how it works. We feel penalized whenever program code or language features are inaccessible to us.

To make a language (or a programming technique) more accessible, books and training materials are always valued. Most of us read several journals each month besides Forth Dimensions in order to have better access to state-of-art practices and techniques.

Tom Saunders of Sigma 3 Engineering in Edmonton (Alberta), Canada requested that FIG members participate in a survey so that every Forth dialect could be briefly outlined and its design goals described (Letters, FD XIII/2). This comment prompts me to question whether there is a way for us to pursue our diverse Forths and diverse programming techniques with any real hope of improving Forth’s commercial standing—which currently seems to be flat growth for a relatively small number of businesses. Undoubtedly, our diverse solutions will also lead to many breakthroughs. But ignorance of these breakthroughs techniques (or innovative Forth dialects) is widespread. How many receive only limited use in a handful of products, if that? Without doubt, the Forth systems comparisons offered by Guy Kelly have helped increase our awareness of the differences between some popular Forths (FD XIII/6).

Based on the comments I scanned, our concerns are becoming focused upon management issues and upon the prominence of the development environment. As we focus on issues such as support and training, we broaden our concept of the total cost of software. Our ability to profit from software will require us to be sensitive to all the issues of producing, deploying, and maintaining software.

Among its credits, Forth natively facilitates fast program development and easy program modification—two of the chief advantages claimed by makers of various development tools. Even without any of the extras that are part of a contemporary development environment, Forth systems are alleged to be perfectly suited to most programming needs. If you can make this claim, fortune may be smiling upon you. Those of us who require database languages with graphical interfaces may disagree.

We’ve also heard strong statements about how much we value our complete understanding of Forth, including the operation of its implementation code. In light of this, consider another of John Wavrik’s comments. Here he questions where the proposed ANS Forth is headed—which he believes is away from Forth’s past openness and low-level accessibility:

“My claim is that Forth has traditionally been a language which allows the user to build major language features. (There is Forth literature discussing variant methods for doing local variables, exception handling, adding object orientation, etc.) Forth has been a toolkit for building application-oriented languages. The ANSI team is heading in the direction of including some important features (local variables, exception handling, etc.) but removing the ability to build such things” (Best of GEnie, FD XIII/5).

I would like to thank everyone who made their thoughts known by submitting them to Forth Dimensions or to GEnie’s Forth RoundTable. Through these forums, we all become better informed about the concerns facing our community.

—Mike Bola
# Volume XIII Index

*Subject index to Forth Dimensions contents published from May '91–April '92. Prepared by Mike Elola.*

| arithmetic operations                          | Forth Dimensions, vol 13, #3, pg 30 |
| blocks within files for source code            | Sixty-formatted Source Code, vol 13, #1, pg 28 |
| chapters, Forth Interest Group                 | Letter, vol 13, #2, pg 31 |
| Lette, vol 13, #3, pg 30                       | conditional compilation |
| Smart Comments & Compiler Words, vol 13, #2, pg 6 | conferences |
| control flow                                   | FORML Thanksgiving, vol 13, #6, pg 38 |
| Universal Control Structures, vol 13, #3, pg 9 | dialects of Forth |
| The Curly Control Structure Set, vol 13, #6, pg 22 | Introduction to Pygmy Forth, vol 13, #2, pg 25 |
| Yerk Comes to the PC, vol 13, #5, pg 6         | Re: Intro. to Pygmy Forth, Letter, vol 13, #4, pg 5 |
| Letter, vol 13, #2, pg 5                       | Best of GEnie, vol 13, #6, pg 32 |
| Re: Intro. to Pygmy Forth, Letter, vol 13, #4, pg 5 | documentation, source code storage within |
| Best of GEnie, vol 13, #6, pg 32               | Sixty-formatted Source Code, vol 13, #1, pg 28 |
| editing source code                            | Add and Delete Screens in PDE, vol 13, #1, pg 23 |
| Add and Delete Screens in PDE, vol 13, #1, pg 23 | Letter, vol 13, #3, pg 30 |
| Letter, vol 13, #3, pg 34                      | Forth Interest Group |
| President's Letter, vol 13, #1, pg 6           | President's Letter, vol 13, #1, pg 6 |
| President's Letter, vol 13, #2, pg 32          | President's Letter, vol 13, #2, pg 32 |
| Letter, vol 13, #1, pg 5                       | President's Letter, vol 13, #1, pg 5 |
| President's Letter, vol 13, #3, pg 23          | President's Letter, vol 13, #3, pg 23 |
| Forth leaders                                  | Best of GEnie, vol 13, #2, pg 33 |
| Best of GEnie, vol 13, #3, pg 38               | Best of GEnie, vol 13, #3, pg 38 |
| New FIG Board Members, vol 13, #6, pg 31       | Hashing |
| hashing                                        | QuikFind String Search, vol 13, #4, pg 21 |
| QuikFind String Search, vol 13, #4, pg 21      | Re: QuikFind String Search, Letter, vol 13, #5, pg 15 |
| interfacing Forth to operating systems         | Sixty-formatted Source Code, vol 13, #1, pg 28 |
| Sixty-formatted Source Code, vol 13, #1, pg 28 | List operations |
| Symbolic Processing, vol 13, #1, pg 7          | Symbolic Processing |
| metacompiling                                  | Symbolic Processing, vol 13, #1, pg 7 |
| eForth—a Portable Forth Model, vol 13, #1, pg 15 | target compiling using a hosted target |
| Re: How Metacompilation Stops the Growth Rate of Forth Programmers, Letter, vol 13, #3, pg 5 | Forth for the 90's, vol 13, #1, pg 12 |
| minimal Forth                                  | eForth—a Portable Forth Model, vol 13, #1, pg 15 |
| Best of GEnie, vol 13, #6, pg 32               | user interface routines |
|                                                   | Menu Words, vol 13, #1, pg 18 |
|                                                   | vocabularies, searching through |
|                                                   | Best of GEnie, vol 13, #1, pg 31 |
| multiprocessor systems                          | Ada Multiprocessor Real-Time Kernel, vol 13, #3, pg 24 |
| object oriented programming                     | Yerk Comes to the PC, vol 13, #5, pg 6 |
| Simple Object-Oriented Forth, vol 13, #5, pg 33 | Object-Oriented Forth, vol 13, #5, pg 23 |
| product reviews and surveys                     | product reviews and surveys |
| Forth Systems Comparisons, vol 13, #6, pg 6     | Forth for the 90's, vol 13, #1, pg 12 |
| Letter, vol 13, #2, pg 5                        | President's Letter, vol 13, #3, pg 23 |
| Letter, vol 13, #3, pg 37                       | Editor, vol 13, #4, pg 4 |
| Letter, vol 13, #4, pg 10                       | Letter, vol 13, #4, pg 10 |
| Best of GEnie, vol 13, #6, pg 32                | Letter, vol 13, #5, pg 5 |
| promoting Forth                                 | Letter, vol 13, #5, pg 13 |
| Forth for the 90's, vol 13, #1, pg 12            | real-time control |
| President's Letter, vol 13, #1, pg 6            | Ada Multiprocessor Real-Time Kernel, vol 13, #3, pg 24 |
| Editor, vol 13, #4, pg 4                        | Neural Network Words, vol 13, #2, pg 9 |
| Letter, vol 13, #4, pg 10                       | Universal Control Structures, vol 13, #3, pg 9 |
| President's Letter, vol 13, #4, pg 26           | Sorting algorithms |
| Letter, vol 13, #5, pg 5                        | Comboort in Forth, vol 13, #4, pg 6 |
| Letter, vol 13, #5, pg 13                       | stack operations |
| New Stack Tools, vol 13, #4, pg 13              | New Stack Tools, vol 13, #4, pg 13 |
| standards, dpANS Forth                         | Best of GEnie, vol 13, #1, pg 31 |
| Best of GEnie, vol 13, #1, pg 31                | Best of GEnie, vol 13, #5, pg 19 |
| Best of GEnie, vol 13, #6, pg 32                | Best of GEnie, vol 13, #6, pg 32 |
| strings                                        | QuikFind String Search, vol 13, #4, pg 21 |
| Symbolic Processing, vol 13, #1, pg 7           | Symbolic Processing, vol 13, #1, pg 7 |
| target compiling using a hosted target          | Forth for the 90's, vol 13, #1, pg 12 |
| Forth for the 90's, vol 13, #1, pg 15            | eForth—a Portable Forth Model, vol 13, #1, pg 15 |
| user interface routines                        | Menu Words, vol 13, #1, pg 18 |
| Best of GEnie, vol 13, #1, pg 31                | vocabularies, searching through |
|                                                   | Best of GEnie, vol 13, #1, pg 31 |

*Forth Dimensions*
Transcendental Compilation

Conducted by Russell L. Harris
Houston, Texas

Among the things which make Forth unique among computer languages is the process of metacompilation. Also known by the terms target compilation and cross-compilation, metacompilation is, in simplest terms, a process by which an existing Forth system is used to generate a second, tailor-made Forth system. In this respect, metacompilation transcends the usual process of compilation. The new system may be a complete development environment, itself capable of metacompilation; it may be a ROMable application, having only the barest essentials to accomplish a specific and limited task; it may be an end-user application, with support for terminal and disk I/O, but without editor, assembler, and compiler. The new system may run on a machine identical to the development system on which the metacompilation takes place; it may run on a machine with word size, instruction set, and resources quite different from those of the development system; it may run from ROM on an embedded single-board computer. Whatever the case, metacompilation enables the programmer to create the new system with a minimum expenditure of time and effort, while giving him a degree of control he otherwise would have only in assembly language.

Daily association with Forth devotees via a local telephone call is an experience you shouldn’t pass up.

The Emperor's New Clothes

Before proceeding with our discussion of metacompilation, it is necessary that several concepts be explained and that a number of terms be carefully defined. The matter of nomenclature is complicated by two factors. First, everyone seems to have his own name for a given item. Thus, what I call a nucleus you may call a kernel. Secondly, there’s always someone trying to get rich by robbing others, specifically, by getting the government to hold a gun to everyone else’s head while he, the robber baron, loots their pockets. If you believe in the non-entity commonly termed “intellectual property,” be sure to promptly send me a substantial fee before proceeding further in this tutorial series; otherwise, I will be forced to dispatch a team of thugs with instructions to repossess my “property.” (Those of you who have not bowed the knee to the idol of “intellectual property” may, with my blessing, proceed without charge.)

I plan to publish a paper on the subject of “intellectual property”; meanwhile, you might wish to visit the children’s section of your local library and read again the faerie tale “The Emperor’s New Clothes.” If you care to research the matter of “intellectual property,” I suggest you begin with the treatise entitled The Law, first published in 1850, authored by the Frenchman, Frederic Bastiat (1801–1850).

Nomenclature

Compilation is simply the process of writing to a dictionary. Compilation is a routine occurrence in Forth development environments, and also takes place in some Forth applications. Traditionally, on a disk-based Forth development system, the bootstrap loader or operating system brings up a small Forth nucleus of approximately 8K bytes. This nucleus then compiles or “loads” the balance of the Forth system, including an application, if any.

Forth words are typically classified into categories, much as routines in C are grouped into libraries. Categories outside the nucleus are termed electives. The set of electives to be loaded varies with the Forth implementation, the preferences of the user, and the requirements of the application, if any. When memory is limited, one need load only those electives necessary to support the application. Electives commonly loaded include those for printing, editing, and disk operations, in addition to the more basic functions such as clock, calendar, and double-length arithmetic.

The process of loading electives and applications is nothing other than compilation. Note, however, that loading the nucleus is not properly termed compilation: the bootstrap loader or operating system simply copies from disk to RAM an executable image. The source blocks which comprise electives and applications contain both high-level and code words. The high-level words are compiled by the colon compiler, while the code words are compiled by the assembler. The resulting executable code is compiled into the dictionary of the system on which the compiler and the assembler are executing; i.e., electives and applications are compiled into the operating environment. Thus, Forth words, both high-level and code, may be executed immediately after they have been compiled.

The meta in metacompilation indicates that the code being compiled is destined for an environment other than the operating environment. Unless the application environment is substantially the same as the development environment, it will not be possible to test metacompiled code within the development environment. Even if the development system and the application hardware share the same word size and instruction set, the complement and physical addresses of memory and peripherals may differ between the two systems.

Rather than attempting to metacompile into the operating environment, one generally sets aside, somewhere on the development system, an area of RAM or disk to receive the executable application code. Once metacompilation is complete, the code may be transferred elsewhere for testing.

In this and future columns, the terms computer system, bandwidth, and machine are synonymous, referring to a
In metacompilation, there are, in principle, two computer systems. The hardware on which the metacompiler runs is termed the development system or host. The term development system is very appropriate for the computer used to write or develop an application, but the term is cumbersome, and I am open for suggestions as to a short yet descriptive name. The hardware on which the metacompiled application is to run is termed the application hardware or target. Again, application hardware is descriptive, but is awkward. Any suggestions? In some cases, the development system and the application hardware are the same machine. In our adventures, the development system will be the IBM-PC and the application hardware will be the 8051-family single-board computer presented in the last column.

A Forth metacompiler is a Forth application which runs on a development system. The metacompiler operates on Forth source code in order to produce executable application code. The source code may be a mixture of high-level and code words. Typically, the source code is read from disk and the application code is compiled to disk, but the application code may be compiled to RAM if the development system has sufficient available memory. Alternatively, the application code may be compiled directly to read/write memory in the application hardware, over a data link (typically, a serial line) connecting the development system and the application hardware.

If the development system and the application hardware have different instruction sets, the term cross-compilation is sometimes used instead of the more general term metacompilation.

To Be Continued...

Let us assume we have a Forth environment which does not include the capability of metacompilation. What must we do in order to add this capability? What problems and conflicts do we face? How do we solve and resolve them? What variations are possible and useful? Subsequent columns will address these matters, as we work our way through development of an 8051-family metacompiler which runs on the IBM-PC.

College Endeavours

A facet of university life I find compelling is the daily association with fellows who are pursuing the same or a similar course of study. A university experience in which one limits himself to attendance at lectures, laboratory sessions, and tests might as well be undertaken by correspondence or by attending night school. It is the opportunity outside the classroom to discuss, to reason, to hone mind against mind, that sets apart the university. In the collegiate environment, you can always find someone who recalls points you failed to note, someone who sees the underlying concept through detail you find impenetrable, someone willing to scrutinize your logic or verify a solution, someone content simply to listen as you think aloud, someone with an alternate perspective and approach to a problem which has you stumped. Conversely, you provide like function for your fellows. It is a give-and-take affair, somewhat like a climbing expedition, in which possession of the secure position is constantly passing from one member to another as progress is made.

While few of us can afford a return, even for a brief period, to full-time academic study, and few of us have employers which foster an interactive academic environment in the workplace, there yet remains a collegiate experience affordable and accessible to almost everyone. For a few dollars a month and a few dollars per hour of connect time, one may gain the potential of daily communication with a large number of individuals pursuing a common goal. I am speaking of the Forth Interest Group (FIG) bulletin board and real time FIG conferences, currently hosted, along with the FIG software library, on the GEnie computer network.

The opportunity of daily association with fellow Forth devotees across the nation, via a local telephone call (Look Ma! No tolls!) is an experience you really shouldn't pass up. The monthly access fee buys unlimited electronic mail, which is great if you need to communicate directly with specific individuals. An hourly charge applies once you move to the Forth "round table," but it is at the round table that you gain access to the FIG community at large.

Once you know your way around GEnie, you can log on, check the FIG bulletin board for new messages in a given category, and log off, all in roughly a minute, so there is little excuse not to look in on a regular basis. You can download from GEnie a freebie utility called Aladdin with which PC users can automate the process, thus eliminating the time normally consumed in handling menus.

I urge readers of this column and every member of FIG to get a GEnie account and join us in an environment of mutual support and exploration. Our sysops have provided a bulletin board category, No. 19, for activity related to the "On the Back Burner" column. Under that category, several topics have been started and others can be added as needed. Readers having questions need to post those questions under the appropriate topic of category 19 and check back frequently, if not daily, for response. Readers knowledgeable in various areas are requested to frequently check topics in which they have expertise and to provide answers wherever possible. Readers having better or alternate solutions to common problems are invited to share their insight with the rest of us.

The gist of it is this: by way of the GEnie computing network, the FIG round table opens the door to interaction on a scale which would otherwise be impossible and on a frequency which would otherwise be prohibitive. Readers of this column who are following the ongoing tutorial need the type of support which only a collegiate environment or a resource such as a nation-wide, local-access bulletin board can provide. To readers who have mastered subjects and techniques covered by this column, the FIG round table offers the opportunity to share insight and to lend a helping hand. Everyone is welcome; everyone is needed. Won't you join us?

R.S.V.P.

Russell Harris is an independent consultant providing engineering, programming, and technical documentation services to a variety of industrial clients. His main interests lie in writing and teaching, and in working with embedded systems in the fields of instrumentation and machine control. He can be reached by phone at 713-461-1618; by fax at 713-461-0081; by mail at 6060 Doddarado Dr., Houston, Texas 77055, or on GEnie (address RUSSELL H).
Call for Papers

13th Annual
Rochester Forth Conference
June 23 – 26, 1993
on
Process Control

Call for deadlines.
Conference includes introductory and advanced seminars
on Forth technology and its application.

Announcing

Definitions: The Institute Newsletter

Call or write for a complimentary copy of the newsletter or
the Journal and learn about our Associates Program.

Forth Institute
70 Elmwood Avenue
Rochester, NY 14611
(716)-235-0168 (716)-328-6426 fax
72050.2111@compuserve.com

Forth Interest Group
P.O. Box 2154
Oakland, CA 94621