$10 Volume XIX, Number 1 May 1997 June

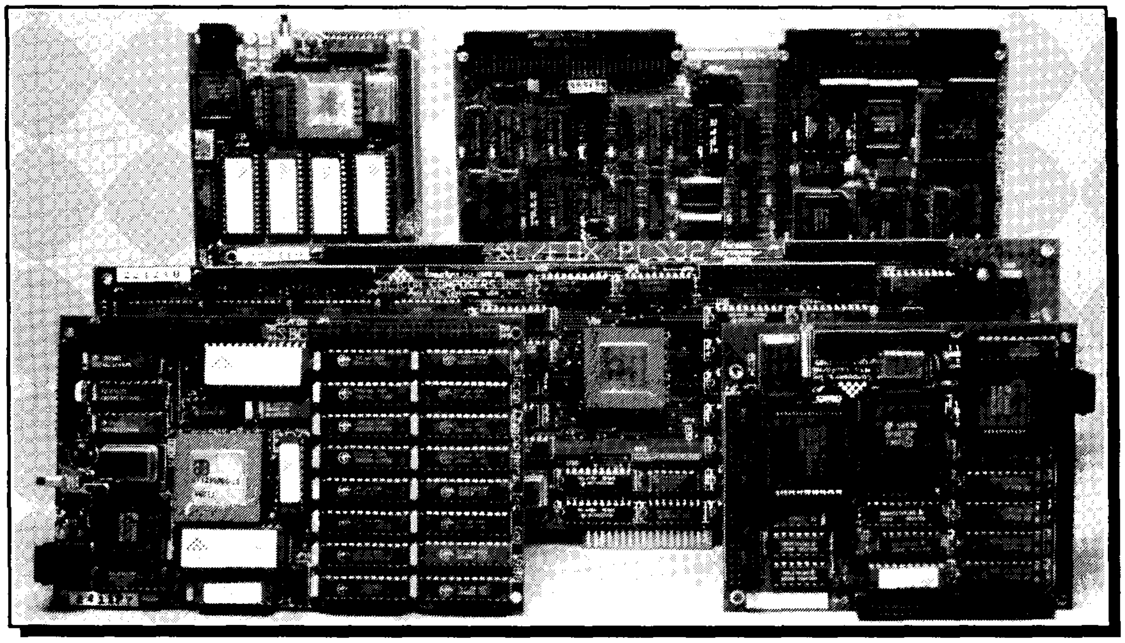
FORTH DIMENSIONS

SILICON COMPOSERS INC

FAST Forth Native-Language Embedded Computers

DUP

>R



C@

R>

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

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

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

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

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

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

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

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

For additional product information and OEM pricing, please contact us at:
SILICON COMPOSERS INC 655 W. Evelyn Ave. #7, Mountain View, CA 94041 (415) 961-8778

EDITORIAL

Smooth Integration

Welcome to a new volume of *Forth Dimensions*, and to a new design. Magazines remake themselves, now and then, for many reasons. In our case, it was time to remind ourselves of the dynamism that change itself can bring. Forth is good at adapting to new situations, and so should we also strive for adaptability as a character trait.

We think the new design is easy on the eyes, but makes a stronger statement on the page. The new display type (Myriad) is easier to read, leaner and much less strident than the tired Helvetica variants we used. The new typeface for body text (Stone Serif) performs better at small point sizes, and fits more text on the page. This will enable us to conserve pages while retaining the same quantity of content or, equally, to provide more content at no additional expense in paper, printing, and postage. Of course, to expand content, we must convince more of you that your written contributions are both welcome and needed!

An early respondee to our recent on-line call for authors is Neil Bawd, whose work has been well-received at several FORML Conferences (this year's event—for the first time—is *not* being held on the U.S.A. holiday of Thanksgiving; see back cover). Other plans are in development, and we encourage you to consider how your contribution might add value to our pages. We welcome participation—in fact, we thrive on it!

A flexible design is one thing; in the professional arena, of course, adaptability means fulfilling a job's requirements in ways appropriate to the job and to the satisfaction of management, a customer, or both. Sometimes that means falling short of an imagined ideal, or taking an approach we would not choose if we were working alone, with unlimited time and the resources of our own choosing. At other times, we can argue convincingly for a more-elegant solution, if we have the rhetoric, the facts, and the kindly disposition of the powers-that-be—and if the schedule permits. Or maybe it just takes the nerve to try John Nangreaves' tactic (see his article, "A Forth Memoir"), which seemed borderline-smart-aleck on first reading, but it apparently worked for him.

The tradeoffs we make for a paying job differ from those we make (or refuse to make) on amateur projects. Paying jobs have budgets and deadlines, and usually involve working with other individuals, with their preconceptions and preferences. Sometimes, instead of insisting, "But we can use Forth to do all that," the political response is, "Sure, I can do this part most efficiently in Forth, and it will be no problem to interface to the rest of the system. Here's how..." Then let the smooth integration of your Forth code, and its reliability and performance, inspire management to take a second look and ask, "What additional pieces can we do in Forth?"

But forging ahead in this work isn't always a matter of convincing non-believers. Working for a Forth company or department can mean conforming to coding styles that might seem foreign to us, even cumbersome when we just want to produce working code. Sometimes it means hammering out compromise with fellow users who don't share the same tradeoff preferences, which is how standards are conceived and one reason they are so difficult to arrive at.

A working environment doesn't offer the luxury of endless philosophical debate. We have to produce working code and deliverable product; to get lost in the nuances of a single battle is to lose the war. Fortunately, the Forth approach to writing programs is conducive to this environment: its interactive write, test, revise cycle means that functioning code that meets specs—and compromises where compromise is required—can be produced in less time, with fewer programmers and fewer demands on resources. Indeed, one commercial interest in Europe recently told me that, while the user-group factions seems to suffer from perennial gloom, his business quietly delivers products regularly to a significant and profitable customer base.

Marlin Ouverson
editor@forth.org

Forth Dimensions
Volume XIX, Number 1
May 1997 June

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Trace Carter

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 $45 per year ($53 Canada/Mexico, $60 overseas air). For membership, change of address, and to submit items for publication, the address is:

Forth Interest Group
100 Dolores Street, suite 183
Carmel, California 93923
Administrative offices:
408-37-FORTH Fax: 408-373-2845

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. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

FORTH DIMENSIONS (ISSN 0884-0822) is published bimonthly for $45/53/60 per year by Forth Interest Group at 1340 Munras Avenue, Suite 314, Monterey CA 93940. Periodicals postage rates paid at Monterey CA and at additional mailing offices.

POSTMASTER: Send address changes to FORTH DIMENSIONS, 100 Dolores Street, Suite 183, Carmel CA 93923-8665.

Figure One.

```
FORTH-WORDLIST CONSTANT 4TH   ( A convenient abbreviation. )
: GENERAL  4TH 1 SET-ORDER   4TH SET-CURRENT ;      ( -- )

WORDLIST CONSTANT IMP        ( For implementation words. )
: PRIVATE  4TH IMP 2 SET-ORDER   IMP SET-CURRENT ; ( -- )
: PUBLIC   4TH IMP 2 SET-ORDER   4TH SET-CURRENT ; ( -- )
```

Figure Two.

```
: GENERAL ONLY FORTH ( ALSO ) DEFINITIONS ;
: PRIVATE ONLY FORTH ALSO IMPLEMENTATION DEFINITIONS ;
: PUBLIC ONLY FORTH ALSO IMPLEMENTAT97-05-17
ALSO FORTH DEFINITIONS PREVIOUS ;
```

Figure Three.

```
PRIVATE
    VARIABLE SEED
PUBLIC
    : SRAND SEED ! ;  1 SRAND              ( n -- )
PRIVATE
    314748364721 CONSTANT MULTIPLIER
    : SCRAMBLE   MULTIPLIER *  1+ ;        ( n -- n' )
PUBLIC
    : RAND SEED @ SCRAMBLE DUP SEED ! ;  ( -- n )
GENERAL
```

Figure Four.

```
PRIVATE ORDER
Search Order (first to last): IMPLEMENTATION  FORTH
Current: IMPLEMENTATION

PUBLIC ORDER
Search Order (first to last): IMPLEMENTATION  FORTH
Current: FORTH

GENERAL ORDER
Search Order (first to last): FORTH
Current: FORTH
or
Search Order (first to last): FORTH  FORTH
Current: FORTH

PRIVATE WORDS
SCRAMBLE        MULTIPLIER      SEED

PUBLIC WORDS
SCRAMBLE        MULTIPLIER      SEED

GENERAL WORDS
RAND            SRAND           PUBLIC          PRIVATE
GENERAL         IMPLEMENTATION  etcetera
```

PRIVATE ... PUBLIC ... GENERAL

In "Yet Another Modest Proposal," (*FD* XVIII/6, 1997), Richard Astle proposes patching the dictionary structure to hide words used for implementation from the end user.

In the early 1980s, Val Shorre proposed INTERNAL ... EXTERNAL ... MODULE for what Dr. Astle calls PRIVATE ... PUBLIC ... END-MODULE.

Dr. Shorre did this like Dr. Astle, by patching the dictionary structure.

Patching the dictionary structure is not portable, and often is not possible.

Standard Forth recognizes what Astle and Shorre have done as *search orders*, and can define them accordingly. As a search order, GENERAL is a better name than MODULE or END-MODULE.

Portable standard code is given in Figure One. This code can be modified for more than one implementation wordlist.

If ONLY ... ALSO is used, first define vocabulary IMPLEMENTATION and then the words in Figure Two.

An elaborated mini-example is given in Figure Three.

In GENERAL, the implementation words won't be available. Any names used in the implementation that were previously defined are still valid, with their original meaning.

Some reports appear in Figure Four.

P.S. In the Forth Scientific Library, IMP, GENERAL, PUBLIC, and PRIVATE are named hidden-wordlist, Reset-Search-Order, Public:, and Private:, but otherwise with identical definitions.

Wil Baden • Costa Mesa, California
wilbaden@netcom.com

Richard Astle replies:

1. I should have been more careful. Although I took pains to indicate—by citing other languages (Modula-2, Ada, etc.)—that the MODULE idea was not my own, I was not aware of Val Shorre's proposal for Forth, or if I was I had forgotten it. My first exposure to the technique of patching the dictionary structure came from Bob LaQuey in the early 80s, who used it in an implementation of a code overlay mechanism in an Apple figForth. If the dictionary

Continued on page 19

OFFICE NEWS

As usual, a lot is happening here at the Administrative and Sales Office. One of the most exciting milestones is that, finally, much of what needs to be done here is becoming routine! There is a definite feeling of security and comfort when things are no longer being done for the first time. We now routinely renew memberships, add new members, fill orders and, yes, even bulk mail *Forth Dimensions* to you! What relief it was when you all started reporting that you really did receive your March/April issue—Julie and I celebrated!

As you probably realize, much goes on behind the scenes to enable an oganization to keep running. Sometimes we make mistakes; your communication to us when you spot these and bring them to our attention is greatly valued. Our business here at the Administrative and Sales Office, in addition to the obvious, is one of communication and support for you, the members.

In this issue, you'll find a new advertiser, Kevin Martin of Management Recruiters. He's been in the tech business for 12 years, and now is working to put the right people and the right positions together. We hope he can be of help to you. And thank you, Kevin, for your support of the Forth Interest Group and *Forth Dimensions*.

Let us know what we can do to better serve your needs. Cheers 'til next time!

Trace Carter
Forth Interest Group, Administrative & Sales Office
100 Dolores Street, Suite 183
Carmel, California 93923 U.S.A.
408-373-6784 (voice)
408-373-2845 (fax)

Trace Carter • Monterey, California
office@forth.org

CHAPTER NEWS

Taking stock:

Southern Ontario FIG Chapter

Minutes of Meeting: Saturday April 5, 1997
Ryerson Polytechnic University

2:00	Informal conversation
3:00	Meeting convened
5:00	Meeting adjourned

After some preliminary soul-searching, the following points were discussed:

Objectives
Suggested objectives were: news/information about Forth-related issues; networking (some of us are contractors or consultants); peer review of our work and ideas; education; resource pool (for information, and possibly programmers); social event for members.

Chapter Coordinator & FIG Liason
Robert McDonald will do this for the next year

Library
Its status and fate were questioned: the files are on disk at McMaster University; Brad Rodriguez has a set of diskettes of the library, and indicated that the entire collection is a subset of the software available on ftp.forth.org, so it is redundant; a somewhat less up-to-date copy of the forth.org collection is available on CD-ROM from Mountain View Press.

Meeting Frequency And Location
Robin Ziolkowski suggested that we meet more frequently; it was agreed to meet every second month, on even-numbered months. Meetings will be the first Saturday of the month, unless it falls on a long weekend, when it will move to the second weekend. A speaker will be scheduled for every second meeting, other meetings will be open discussion sessions on Forth-related topics. Robert McDonald will make a list of meeting dates for the next year. Permission to hold regular meetings at Ryerson has not been received yet; Robert McDonald will contact our Ryerson liason with the proposed schedule. Ken Kupisz will investigate the possibility of meeting at the Ontario Hydro office building; other potential locations suggested included libraries, other schools, and newspaper auditoriums (similar to that of the Hamilton *Spectator*).

Chapter Web Page
Nicholas Solntseff will set up a web page for the chapter, and a link will be requested from www.forth.org. Meeting schedule will be posted on the web page.

Programs
Member presentations; presentations by invited guests; tutorials—at various levels, for members and guests; open days/exhibitions (we should participate in local computer fairs, etc., space may be available free of charge for groups like FIG). The suggestion that we go to dinner together after meetings met with general approval; details to be arranged after each meeting. Look out for parking lots that charge a premium if you stay late! We should be providing speakers to other groups.

Meeting Notices
It was noted that we should ensure that required approvals are obtained for posters placed at Ryerson, so they will not be removed. Meetings notices should be published in *Toronto Computes* and *The Computer Paper*. There are program-specific

campus news groups where meetings could be announced. John Verne says an auto-posting program exists that could be used to post meeting notices on the Usenet. Toronto computer user groups should be notified of meetings.

Other

Members are interested in holding a joint meeting with the Detroit chapter of FIG. It was noted that London (Ontario) would be a good site, as it is halfway from Toronto to Detroit. The Detroit chapter has not yet been contacted about this. The Rochester Forth Conference is June 25–28, in Rochester this year.

It was suggested that the August meeting be a BBQ/social event. This met with approval, and several members offered to host the event. No details were worked out.

Next Meeting

Speaker: Robin Ziolkowski
Topic: Fuzzy Logic

NEWS FROM EUROPE

In appreciation of the time Claus Vogt (clv@clvpoint.forth-ev.de) and Wolf Wejgaard spent sending me their notes from the recent Forth conference in Ludwigshafen, I have constructed an English version, and would like to share them with others who might be interested. —HV

Our European Forth friends have been getting my short reports about the Silicon Valley Forth Interest Group Chapter's monthly meetings, and now I have a report back from Germany, about the recent Forth *Tagung* in Ludwigshafen. I think it is reassuring to know that Forth lives and that Forthers work on similar projects around the world, so let me pass forth my best interpretation of what I have received.

The meeting for the German Forth Society (April 25–27) was organized by Ewald Rieger, who controls robots for a chemical company in Ludwigshafen. Contrary to the wishes of higher management, these robots do not speak other languages, but do their chemical analyses in Forth, since there are no better alternatives.

Ewald was one of the more than 20 speakers. More than 30 people attended overall. Now, the other speakers...

Klaus Schleisiek was there and showed his ultimate multitasker.

Bernd Paysan brought a Visual Forth that is a complement to his BigForth, with which he can operate similarly to Visual Basic or Delphi. Bernd Paysan and Jens Wilke also told about their last-minute "red-eye" project: porting Gforth to MISC (Minimal Instruction Set Computer) in two days. (There is a full story in the first-quarter-1997 issue of *Vierte Dimension.**)

Stefan Lange and Egmont Woitzel both spoke about the theme of working Forth into modern Windows applications, without excessive development efforts. Stefan showed various possibilities of tying Forth into Delphi. Egmont took Win32 as the basis and demonstrated how Forth functions can be addressed in other programming systems (DLLs).

Ulrich Hoffmann has ported Forth onto the PIC (a minimal computer with a few hundred bytes of memory). He also showed the group how Forth exists on the Internet.

Heinrich Moeller develops software for database and management applications for the medical profession. His present product is a "huge" system written in Win32Forth, but he finds it superior to the previous one, which was done in ProForth. Compliments to Tom Zimmer!

Anton Ertl spoke about optimization of stack code, and showed his Gray parser.

Manfred Mahlow has a context-oriented Forth system, in which blocks are used in a clever way to maintain modularity of control.

Arndt Klingelnberg spoke about Forth and the CAN bus.

Wolf Wejgaard (the author of HolonForth) kindled the philosophical discussions about Forth and the future. It is his opinion that the computer scientists *cannot see* Forth, because they are so formally oriented that Forth simply doesn't fit in their picture.

*A sample issue of *Forth Magazin Vierte Dimension* may be ordered from Forth-Gesellschaft,
PF 1110, -85701 Unterschleißheim, Germany
http://www.informatik.uni-kiel.de/~uho/VD

Henry Vinerts • Newark, California
v.vinerts@genie.com

OFF THE NET

University of California – Santa Cruz
Computer Engineering,
Programming Language Proficiency

"Students must show proficiency in some programming language other than C or assembly language. This proficiency can be demonstrated by advanced placement or transfer credit for Pascal, or an approved transfer credit course in languages such as Cobol, Fortran, Lisp, Prolog, C++, Modula2, Forth, or an assembly language different from that of course 121. For students who lack applicable transfer credit, this proficiency can be shown by passing Computer Science 109, 112, or 115 as an upper-division elective, or by doing a substantial research or development project in a different language."

Rochester Forth Conference

RFC:
http://cis.paisley.ac.uk/forth/rfc/index.html

RFC '97:
http://cis.paisley.ac.uk/forth/rfc/rfc97.html

euroFORTH Conference

EuroForth:
http://cis.paisley.ac.uk/forth/euro/index.html

EuroForth '97:
http://cis.paisley.ac.uk/forth/euro/ef97.html

A Forth Memoir

Forth? I had been coding assembler, in hex even, for about a year and a half. Intel, Motorola, and the infamous Z80; mostly multiple-processor boards—some had as many as eight Z80s. I remember the first 68000, and the very not politically correct expression I uttered when I first saw one. I am not a programmer, I'm a technologist; the effect the 68000 had on me was profound. Too many opcodes, too little time. I was the master of the 8085, and the feeling of obsolescence was looming. Typing hex codes into a PC is really no way to get any work done. There had to be a better way.

The next job seemed simple at first: decode a serial stream from an RF receiver, and hand the codes to a serial port. I took the job before they told me the clocks from the transmitters (passive tags) were ±25%, and I discovered they had a tendency to suddenly stop transmitting. A parity bit would have been almost as helpful as a stop bit but, because the tags were already built, I was stuck with making a receiver to suit them.

What I needed was a small computer with a real-time, interactive operating system so I could fool around with the code on the fly and see what worked or, more importantly, what didn't. The thought of all that assembly, compiling, testing, and revising almost drove me to drink. Surely this is a common situation, I thought, so what do big companies do? A little research showed me the two common options, neither of which was appealing: use a desktop, or do it the hard way. A viable solution must exist; this is 1987, after all!

Too many opcodes, too little time.

May the Forth Be With You

The solution was an OEM single-board computer, or SBC, and Vesta Technology (Wheat Ridge, Colorado) had one with my name on it. I owe a big part of my career to Stephen Sarns and Jack Woehr, who told me what I needed to know.

"Read this book." They encouraged me to read Leo Brodie's *Starting Forth*, which seems to have been the standard starting point for aspiring Forth programmers for years.

"Use this board." They sent me all the documentation, software, and boards I needed.

Jack even sent occasional chunks of code to answer questions I sent him. That old 1200 baud modem spent a lot of time on the RCFB (Real-time Control and Forth BBS). I later learned that this is typical in the Forth community: I help you, you help me, they help us, we help them, and everybody benefits.

For a hardware designer, I sure churned out a lot of code. Forth fit almost everything I had to do; I never had to use a different language again, although I did have to learn C to decide I liked Forth better! I never used a PC as an application's engine, either, although I did try, on occasion.

I used several revisions and generations of SBCs, in a wide variety of applications—from cheap and dirty, to big, complicated, and expensive. It got to the point where half the equipment in the shop was either based on or connected to an SBC. In 1988, I had a programmable pseudo-random signal generator with a record feature; an SBC with D/A and A/D; a function generator; and a few K of Forth. Only one on the block. And the projects kept getting bigger, faster, and more complex. I eventually had to stick to hardware-level programming, or there wouldn't be enough hardware to program. So I converted a C programmer... which was not very hard. I did the hardware and hard/soft interface, he did the application, and (with a little help from Vesta) the acceleration took us to critical velocity in no time.

All good things come to a change. I left that position for another, but took my expertise (not to mention my SBCs) with me. I soon had more applications than resources. Time to leave the nest.

The boards I was using just did not fit the application. Too much this, not enough that. So I'll just design my own. The processor decision was the hardest part. I needed all the hardware specifications to fit, but I also needed an operating system. A little research and a few questions turned me to an 8051-type micro and F51. I then found a book and software (through FIG, I might add) on how to make your own 8051 Forth development system. So I did. Thanks to Bill Payne and Henry Neugass and, indirectly, Sandia National Laboratories.

My very own SBC was designed to operate autonomous, remote-sensing systems, including controlling instruments, logging, telemetry, and diagnostics, when needed. It had to be smart, fast, low power, reliable, inexpensive, small, and adaptable. Everything every SBC should be. And it is. And it wasn't really very hard to do, either, although a big chunk of it (the Forth operating and development system software) came in the mail. The operating system needed fine-tuning, as would be expected because of the differences in hardware, and I added a bunch of hardware-support code and some high-level words I was used to. All our current products that have a micro use it.

I am presently working on replacing our SBC with PIC microcontrollers, where it's feasible. Hopefully, the source will port easily (a relative term) to whatever Forth I use for the PIC I choose. Been there, done that, doing it for the last time again. I suppose if technology and demand were not ever increasing, I would be out of a job, or at least bored.

Go Forth...

After dozens of prototypes and systems, I realize I have a story to tell, and this is it. Many of you will find it familiar, if not rhetorical; some entirely alien. That's the magic—if you are reading this, you already know Forth, or are about to learn it.

I have heard stories about dynamic spark curves for car engines and macros for accounting software. Forth works just

John Nangreaves, E.E.T.
Musquodoboit Harbour, Nova Scotia, Canada
johnnang@atcon.com

about everywhere. It's most common, or should I say *visible*, in the scientific and military communities, but it cannot be avoided in modern life; your car may run on Forth, and your microwave almost surely does. Mundane things like space shuttles, satellites, observatories, telecommunications networks, industrial robots, and missiles run on Forth, not to mention high-tech things like your building's environmental control and telephone systems. And a variety of consumer electronics too long to recite (look inside your camera; see that micro?). Used to be that just about anything with an alphanumeric LCD module had a Forth history. You cannot deny its per-unit volume, you just can't see it.

It would be different if Borland had Forth++ or Bill had MS-Forth (rumor has it MS does use Forth, but I haven't seen any discussion traffic). Or if they taught Forth in university. One local university gives it an honorable mention, "For those of you who will work in embedded systems, you will encounter an obscure language called Forth; I've heard good things about it, but..." ANS Forth is a step in the right direction, but mainstream use requires mainstream marketing. Until this happens, Forth will have to be looked for, because it is not going to "start you up" on prime-time TV (did I mention TVs and VCRs use such micros?).

... I did have to learn C to decide I liked Forth better.

For those of you using desktop machines, Forth will get anything you need done running, at least equal to any other language. The support base is the whole Forth community, which seems to have taken up a comfortable residency on the Internet, so there really is no excuse not to use it. Besides, how much time have you spent listening to Muzak on the 1-900-so-called-support line, when you could have browsed comp.lang.forth and probably found your answer without posting the question?

Forth shines like the sun when embedded. I saw a Sun micro running a lift at a mine, autonomously controlling the motor torque directly via sensor feedback, as well as scheduling, logging, weighing... essentially a big, embedded Forth system. PIC micros are going into everything from pens to planes now. Application-specific, or custom-microprocessor and microcontroller-based, boards and systems are popping up everywhere in everything. OEM SBCs still hold the niche between PCs (too big) and custom (too involved). And this little group of dedicated professionals using an "obscure" language are mostly responsible for this embedded revolution. The reason: because we could do what wasn't feasible in other soft environments.

The Forth Interest Group and its members are mostly responsible for keeping Forth alive, although credit ultimately must go to the creators of hardware and software components that make it all tangible. If you have a job to do, they make the tools. I have also found they usually support products, which they often are giving away, better than most that typically are paid for. Pride seems to be a higher level of incentive than money. We choose to use Forth because it does what we need it to do, as well as, if not better than, other environments. We also choose to help each other when we can, because we are a small, but unusually significant, community, and we are content to stay that way.

When someone asked me why I use Forth (in that *certain* tone), I replied simply, "You obviously don't." He walked away puzzled. He later told me he had no idea Forth was so prolific, and he felt it was the exact tool for the job. It seems I compelled him to look it up. It has since become my standard reply to that question.

The author is currently Senior Technologist at Magneto Inductive Systems Ltd. (Correpro Atlantic Ltd.).

FREEWARE & SHAREWARE

Transputer Forth

Looking for an easy way to program a transputer?

Six years ago, a huge amount of money was needed to acquire one of the famous OCCAM development kits for the then much-advertised INMOS transputer family. Why should it always be OCCAM, I thought, why not Forth? There was no transputer Forth available at that time—at least none I was willing to spend money for. So I started my own development kit, in Forth, for the T80x family. I'm now releasing the first version, F-TP 1.00. Here are some of its features:

- Forth-83 with almost all of the ANS Forth core integrated.
- Trigonometric functions,
- Metacompiler written in Turbo-Forth, operating from the DOS host as a cross-metacompiler (Turbo-Forth is a DOS-filesystem-based, 16-bit Forth for IBM compatibles, developed by Marc Petremann and the French Forth Group in 1989 and later).
- Automated metacompilation, also for multisystems.
- Parallel processing, similar to the method employed with OCCAM2, adapted to Forth.
- Multisystem option (several transputer Forth systems kept in the same transputer RAM, with access to the respective stacks from any subsystem).
- DOSKEY emulation (`FORTHKEY`, invoked by hotkeys).
- Collecting characters and strings from screen via hotkeys.
- DEBUG (step-by-step tracing).
- Disassembler `DIS` (also disassembles transputer code fragments residing in transputer RAM, such as OCCAM object code).
- Decompiler `SEE` (integrated disassembler for code definitions).
- Assembler in UPN, fully Forth-like usage, structured (`IF THEN ELSE`, etc.).
- Easy calling of any DOS command or program: `DOS <name>`.
- All source.
- Server from host, written in Turbo-Forth.
- Assembling, `PEEK`ing, `POKE`ing, disassembling, `DUMP`ing (also from server, in case the transputer Forth system breaks down).
- Easy, interactive, menu-driven modification of link adapter port addresses: CONFIG.BAT.
- On-line modification of end-of-system address (increasing

Continued on page 25

Fred Behringer • Muenchen, Germany
behringe@statistik.tu-muenchen.de

ftp.statistik.tu-muenchen.de/incoming/Forth/f-tp-100.exe
Freeware, 900 KB, self-extracting

A Simple Implementation of the Kermit Protocol in Pygmy Forth

Kermit in Pygmy

Kermit is not only the name of Henson Associates Inc.'s famous frog, but is also the name of an extensive, complex, file-transfer and remote-computer-access application—written by Columbia University and/or volunteers, as I understand it—sold and distributed by Columbia University[1]. The name *Kermit* is also used to refer to the family of file-transfer protocols used in Columbia University's Kermit and by many modem programs. As I use the term throughout the rest of this article, it refers to the Kermit protocols, particularly the simplest form, implemented here in Pygmy Forth.

My Forth applications sometimes need to transfer files with other systems via modem. I felt I needed to implement one or more of the following protocols: XMODEM, YMODEM, Kermit, and ZMODEM. The basic XMODEM protocol doesn't have much respect these days, as it does not transmit exact file lengths, but rounds up to the nearest block size (128 bytes). Enhanced versions of XMODEM, such as YMODEM, overcome this. Kermit is well thought of, and is widely available. ZMODEM is usually considered to be the best, although there seem to be arguments between the Kermit and ZMODEM camps as to which is the better, faster protocol under various circumstances. I, personally, found the ZMODEM arguments more persuasive, but I don't really care about minor differences. I wanted something that works reasonably well and is fairly easy to implement. Looking over the Kermit and ZMODEM protocols, I decided that a simple form of Kermit would be slightly easier to program. Later, I hope to implement ZMODEM, as well.

Why Write My Own

You may well ask, why insist on writing a version instead of using an existing product. We have experimented some with other products, and have been severely disappointed. For example, we have had a lot of trouble trying to get PCAnywhere configured correctly under DOS. On some client systems, we were unable to configure it by changing the configuration file in the directory from which PCAnywhere would be run. Instead, we had to resort to a Rube Goldberg (i.e., extremely cumbersome): We had to modify the batch file temporarily (so it would not call the PCAnywhere script), run the application and let it shell out to PCAnywhere, change and save the settings, and again modify the batch file so it would run the script. We never found why we could configure it in some offices without going through this tedious process, but not in most; but it was enough to make us hate PCAnywhere.

Further, while the modem scripts would run under DOS, PCAnywhere was too slow. Also, we could not make the DOS versions of the scripts run under the Windows 95 version of PCAnywhere. Also, it was not free, so we had to fool with getting each client to purchase PCAnywhere. Similarly, the use of Columbia University's Kermit product would have required each customer to purchase a license for it (yes, it is *not* free). Ditto for the Oman Technologies ZMODEM product. So, to hell with the third-party products. We will just write our own and have full control, simpler installation, and a faster user interface. Were this running under Unix/Linux, we would have used the built-in, freely distributable ZMODEM.

Frank Sergeant • San Marcos, Texas
pygmy@pobox.com

The Kermit Protocol

My main guide for the Kermit protocol was *C Programmer's Guide to Serial Communications* by Joe Campbell[2]. Note, there are some errors in Campbell's examples.

Files are transferred in Kermit by exchanging *frames* between the sender and the receiver. Every frame begins with an SOH (start of header) character, and ends with a carriage return. Between them are the following five fields: length, sequence, type, data, and checksum. Each field, except for the data field, is a single character in printable form. The process of converting a number, say for the length field, into a character, is called *character-ization* and is done by adding the value of the space character to the number. Thus, the number zero becomes $20 and prints as a space. The number thirty-seven (i.e., $25) becomes $45 and prints as the letter "E," and so forth. I use the words CHAR and UNCHAR to convert between the two formats. Note that this requirement of fitting numbers into a single byte as a printable character limits the numbers to the range 0–94, and limits the size of frames that can be transmitted. Various extensions to the basic Kermit protocol allow longer frames, but not the simple form described here.

The frame types used are S to initiate a file-transfer session, F to send a filename, D to send a data frame, Z to indicate end of file, B to indicate end of transmission, E to indicate a fatal error, A to send file attributes (we do not use this one), Y to ack a frame, and N to nak a frame. No single bytes are exchanged between the sender and receiver, only whole frames. The length field indicates the number of bytes to follow the length field, up to and including the checksum field, but not the carriage return.

To transfer a file, the sender waits for an N-frame (a negative acknowledgment), then sends an S-frame to tell the receiver what values it wants to use for various protocol parameters, such as maximum frame length, the repeat character, the escape character, etc. The receiver then sends a Y-frame (an acknowledgment), with its preferred values for these parameters. The word COMPROMISE takes the more conservative value for each of the parameters. This single exchange of frames sets the protocol parameters for the session. In the S-frame and its Y-frame, the values of the parameters are sent in a fixed order in the frames' data fields.

Kermit has two major ways of converting bytes before transmitting them. One is the character-ization already mentioned, for converting numbers. The other is *controlification*, to flip a bit in a control character to turn it into a printable character. Numbers, as such, are expected in the length, sequence, and checksum fields, but never in the data field (except during the initial S- and Y-frames that establish the protocol parameters). Kermit would allow for transmitting seven-bit data bytes by escaping, with an ampersand, each byte with a high bit set and then clearing that high bit; but we do not do this. We assume the availability of an eight-bit channel. Kermit does insist, though, on escaping control characters (with "#"), and at least some implementations insist on compressing data by run-length encoding repeated bytes. # and ~ characters, when appearing in the data as themselves, must be escaped. Thus a single # would appear in the data field as ##.

To keep things simple, we never compress the data when we are sending a file. Unfortunately, not all implementations of Kermit respect the request not to use repeat counts. Therefore, we must be prepared to handle compressed data when receiving a file. (Since our main purpose in using Kermit is to transmit zip files, it doesn't look like we would gain much speed by compressing the data we send.)

Kermit allows various types of checksums, ranging from a single byte to three bytes. Our main use will be to transmit zipped files, with the zip file itself providing an additional level of data integrity verification with its 32-bit CRC; therefore, I am content to use a one-byte checksum, the simplest form Kermit allows.

The V-frame

We invent a special input frame and assign it type V. A V-frame is never actually sent. Instead, whenever a timeout occurs, `KSER-IN` terminates (*and* terminates its caller `GETFRAME`) and returns a V-frame. This way, a timeout is not a special case, but just an ordinary "frame."

What Else Can Go Wrong

In addition to a timeout, a frame with a bad checksum might be received. In this case, we send an N-frame to alert the sender to try again. If we are receiving and a frame is lost, the V-frame alerts us to the missing frame and we, again, send an N-frame to request a repeat. When sending, if a frame is lost or its Y-frame is lost, either a timeout or receipt of an N-frame alerts us to the need to resend that frame. When we receive a duplicate frame (perhaps because our Y-frame never reached the sender), we basically ignore it. However, we do acknowledge it, so the sender will be free to continue. The progress of a file transfer is indicated on screen by printing a dot for each frame transferred. Our code will wait forever, attempting a transfer, unless it is canceled by the user or it receives an E-frame, indicating a fatal error.

The Environment

This implementation expects the `SER-IN`, `SER-OUT`, and related words that allow transmitting to, and receiving from, the serial port connected to a modem. "The PC Serial Port in Forth"[3] describes one approach to handling the serial port on a PC, using interrupts.

My current approach does not use interrupts. Instead, it uses the new multi-tasking ability of the upcoming Pygmy version 1.5. The serial input is serviced by a separate task, thus greatly simplifying the serial port code.

The Kermit code presented here should work, regardless of which form of serial handling is used.

How to Transfer Files

Assuming the modem connection has been made and the other end is expecting to send or receive the file using Kermit, type *filename.zip* `SEND` to send a file, or type `RECEIVE` to receive a file. The `RECEIVE` routine is capable of receiving multiple files. The `SEND` routine transmits a single file. Since Pygmy 1.5 accepts instructions from the command line, you could type

```
PYGMY " filename.zip" SEND BYE
```

on the DOS command line, or in a batch file, etc. In our application, we have a menu and a terminal mode, with PgUp and PgDn keys invoking the `SEND` and `RECEIVE` routines. You can see a clue as to how we use this in the definition of `KSER-IN` (block 12007), where a user abort of the file transfer executes the `DEFER`'d word `MYMENU` to put the user back at the application menu.

Conclusion

This code has been used successfully at a number of client sites but, remember, it uses only a very basic version of the Kermit protocol. It very well could have problems transferring between certain sites. If you try it, please let me know your results. For your downloading convenience, I have zipped the Kermit source and shadow blocks, along with a copy of this article, and a preliminary version of PYGMY.COM that contains the multi-tasking version of the serial port words, and have placed this on my web site.

References

[1] "Kermit: A File-transfer Protocol for Universities," parts 1 and 2. Frank da Cruz and Bill Catchings, June and July, 1984, *BYTE*.

[2] *C Programmer's Guide to Serial Communications*, Joe Campbell. Howard W. Sams & Company, 1987, ISBN 0-672-22584-0, pp. 98–113. (Note: there are some errors in Campbell's examples.)

[3] "XT Corner: The PC Serial Port in Forth," Frank Sergeant. *The Computer Journal*, issue 79, Fall 1996, pp. 5–10. My web page (http://www.eskimo.com/~pygmy) contains a link to *TCJ*'s web page.

Code appears in the next issue...

Frank Sergeant, on the "thirty-year plan," received his Master of Science in Computer Science from Southwest Texas State University. The degree and his 4.0 GPA were hard won. His thesis was entitled "Calculating the Crust of a Set of Points in C++," and has nothing to do with Forth except to help illustrate that computational geometry is easier in Forth than in C++.

http://www.eskimo.com/~pygmy/forth/kermit.zip

Control Structure Salad Bar

Transportable Control Structures

Standard Structures

The extensibility of the Forth language has long been one of its most popular features. Now ANS Forth has set in print one aspect of this concept which many of us have enjoyed using for years: the idea of creating new types of control structure words without writing any new words in assembler. Obviously, if one is to perform a transfer of control (such as the conditional forward branch in an `IF`), there will some machine code involved in changing the value of the interpreter pointer. Typically, the compiler directive (`IF`, in this case) will compile a reference to a word written in machine language to accomplish this. Potentially, every similar control structure directive could be written in machine language. However, as the standard points out, it is not necessary to do this for all of them. Some of these words have some behavior in common. For example, both `IF` and `WHILE` perform a conditional forward branch at run time. Their common run-time behavior can be factored out and shared.

The ANSI Forth Standard proposes a minimum wordset of `IF, THEN, BEGIN, AGAIN,` and `UNTIL` (very old friends) plus (these are a little newer) `AHEAD` (unconditional forward branch), `CS-PICK` (control stack pick), and `CS-ROLL` (control stack roll). With these tools, we can build all the structures we are used to (such as `WHILE` and `REPEAT`), plus whatever new ones occur to us. The standard goes on to show how to implement one type of `CASE` statement with these tools. I think it is fair to say that many Forth programmers have written their own version of the `CASE` statement, so creating yet another `CASE` could be a stale topic for the more advanced programmers. However, even those who have written Forth compilers might benefit from an occasional reminder of how much freedom and flexibility this language gives us. I recently had an occasion to add a feature to the `CASE` statement which is a part of the Open Firmware system we are putting in the PowerPC systems here at Motorola Computer Group in Tempe, Arizona.

The point of this discussion is that I was able to add the modification quite easily, using existing assembler routines, because the system is designed to allow that. I knew I did not want to spend a lot of time writing new assembler routines. I also knew that, when using other Forth systems in the past I have, on a couple of occasions, taken advantage of this concept to add `CASE` statements to systems which did not have them. In each of these instances, a `CASE` statement was needed in a Forth environment which had no `CASE` and was running on a microprocessor for which we had not implemented machine language primitives for `CASE`. Furthermore, it was not considered important enough to learn the new processors instruction set just to add this one feature.

Because I had been able to add the `CASE` structure by compiling references to existing assembler routines for control structures in other systems, I figured I should try that this time. What I did not realize, until later, was that ANS Forth actually specifies this feature of Forth's extensible nature. (Please note that the standard's example `CASE` is only for demonstrating the reuse of control structure words. It is not recommended for actual use in a system. Most systems would actually use a more efficient implementation.)

Randy Leberknight • Tempe, Arizona
randyl@phx.mcd.mot.com

The Case for CASE

First, let's provide some background into the history of the `CASE` statement in Forth. In the good old days, the entire Forth kernel fit in eight 1K blocks. Real programmers invented whatever control structures they needed, if and when they needed them, and nary a byte was wasted. The thought of having a `CASE` control structure built into every system was considered heretical (if it was considered at all), for two reasons. The first was that a given program might not need a `CASE` statement. Making the standard system carry around the baggage for one, just so some other programmer could use it in another program, was considered a terrible waste.

The second reason for not automatically including a `CASE` statement in every system was that the statement might not be optimized for the particular situation in which it was to be used. Sometimes we need to optimize for speed, other times we need to optimize to minimize memory usage. Some situations call for special handling of default cases, such as not allowing them at all, or expecting the selector to be consumed by a programmer-supplied default action.

Some years have passed, and typical computer resources are much greater now. In desktop systems, we have on-chip caches with more memory than whole computer systems used to have, and the common units of measurement for processor speed are megahertz and MIPS. I assure you, this does not mean we can forget the whole concept of efficiency, and perhaps I can take on that issue at another time. Nevertheless, it does mean that, except in certain special cases, we don't worry so much about memory usage or CPU cycles. Instead, these days we worry more about how long it takes programmers to write and, especially, maintain programs. Therefore, we in-

Figure One.

```
: sample ( selector -- )
   case
     70 of  do-this-stuff endof
     80 of  do-this-stuff endof
     90 of  do-this-stuff endof
     99 of  do-this-stuff endof

     100 of do-some-other-stuff endof
     103 of do-quite-a-different-thing endof
   endcase
;
```

Figure Two.

```
: sample ( selector -- )
   case
     70 over = IF drop   do-this-stuff endof
     80 over = IF drop   do-this-stuff endof
     90 over = IF drop   do-this-stuff endof
     99 over = IF drop   do-this-stuff endof

     100 of do-some-other-stuff endof
     103 of do-quite-a-different-thing endof
    endcase
;
```

Figure Three.

```
: sample ( selector -- )
   case
     70 over =          ( selector flag )
     over 80 = or ( selector flag )
     over 90 = or ( selector flag )
     over 99 = or ( selector flag )
     IF drop   do-this-stuff endof

     100 of do-some-other-stuff endof
     103 of do-quite-a-different-thing
endof
   endcase
;
```

Figure Four.

```
\ set[  marks the beginning of a set of numbers to be
\ compared as a group in a CASE statement. If any of
\ the group match the selector, the following case
\ path is taken.

\ return false as a seed for ORing Booleans:
: set[  ( -- false )    false ;

\ in-set compiles a test for each member of the group,
\ and ORs the test result with an existing flag.
: in-set ( selector1 flag1 value -- selector1 flag2 )
   2 pick  = or
;

: sample ( selector -- )
   case
     set[  70 in-set
         80 in-set
         90 in-set
         99 in-set
     IF drop   do-this-stuff endof

     100 of do-some-other-stuff endof
     103 of do-quite-a-different-thing endof
   endcase
;
```

clude a CASE statement which is easy to use and easy to read. If it needs a feature added, we add it at the highest level possible and go on with our lives. As always, we stay aware of our needs and, when we need blinding speed, we may switch to assembler after careful analysis of the problem.

Having described the background, let's consider what came up in this case. It seems that some programmers here had some code which used CASE statements, but the same behavior was needed for several cases. This produced code like that in Figure One.

They were bothered by the repetition involved here, and I didn't really blame them. After a little discussion, one person suggested he had used the concept of a *set* in the past, so I decided to have a try at implementing that idea. The first try was easy, short, and had a really ugly syntax. I include it here not as an example of a nice structural syntax (it isn't nice), but as an example of how flexible these structures really are. In particular, it demonstrates that OF is syntactically equivalent to an IF with a DROP in the body to discard the selector if this path is taken. ENDOF is similar to an ELSE whose target is the ENDCASE. In our Open Firmware system, the primitives are done in assembler for speed, but the behaviors are as I describe them. That means the above phrase could be replaced with the contents of Figure Two.

Therefore, we could combine all the Booleans into one IF clause by using OR to combine flags (Figure Three).

Now make a word which handles the housekeeping for the stack effects (Figure Four).

Add what some may call syntactic sugar (Figure Five) and we see the etymology of the ugly in-set syntax. While we are not going to keep the ugly syntax, we did learn a lot about mixing and matching control structure words along the way.

The final change was to remove the need for in-set. The easy way is to require the programmer to tell how many items will be tested in the set. This has the same problem as in-set: we are asking the programmer to do work the compiler could do instead. If we intend to leave it all up to the programmer, just let the programmer say (n) over = IF drop. Instead, I decided to figure out how many items were on the stack. Just let set[call depth, and pass the stack depth to] set-of. Unfortunately, we cannot just call depth in set[and expect to retrieve the value from the data stack later, because we will not know how far down on the stack the depth value is. That's why we are calling depth in the first place, to find out how many parameters are passed to] set-of.

One way of sneaking information between two routines is to hide it on the return stack. This is not something I generally recommend, because it's both ugly and dangerous. In particular, it violates one of the most basic factoring rules, that words should be able to function as standalone units. Depending on data passed via the return stack binds these words so that they must always be used as a pair. However, control structure words are generally meant to be used that way. Now, set[

becomes as shown in Figure Six.

Note that we steal the return address which set[needs, put the data we are passing on the return stack, and then restore set['s return address just prior to exiting set[.

All the routines are more complicated now. Ned Conklin (one of the founders of Forth, Inc.) once told me that "complexity is conserved." Which is to say that effort spent in understanding problems leads to simpler code later. If we don't spend energy understanding the problem, we spend energy coding around our ignorance. It also applies in a different way here. We remove the complexity of in-set from the programmer's view, and the complexity reappears (as does in-set) in the internals. The new version of in-set will be called in a loop compiled by] set-of (Figure Seven-a).

Set-of has to compile so much that we factor it out into (setof), as in Figure Seven-b.

I added a default phrase for the test, and put in visible stubs for the sample behaviors. The final usage looks like Figure Eight.

Here are the results of running the test. (Note that we can interpret Do loops at the Open Firmware command line. I like that.)

Figure Five.

```
\ ] set-of terminates a set of numbers to be compared as a
\ group in a CASE statement.
: ] set-of ( -- )
         ( runtime effect: selector flag -- selector | null )
  postpone if   postpone drop  ;  immediate


: sample ( selector -- )
  case
    set[   70 in-set
           80 in-set
           90 in-set
           99 in-set
    ] set-of  do-this-stuff endof

    100 of do-some-other-stuff endof
    103 of do-quite-a-different-thing endof
  endcase
;
```

Figure Six.

```
: set[  ( selector -- ) ( r: selector depth )
  r>            \ get ret-addr  ( selector ret-addr  ( r: 1-short )
  swap >r       \  save selector on return stack  ( ret-addr ) ( r:selector )
  depth >r      \  save stack depth   ( ret-addr ) ( r: selector stack-depth )
  >r            \ restore ret-addr  (   )  ( r: selector stack-depth ret-addr)
;                                   (   )  ( r: selector stack-depth )
```

Figure Seven-a.

```
: in-set ( xu...x1  selector1 flag1 -- xu...x2 selector1 flag2 )
   over         ( xu...x1 selector1 flag1 selector1 )
   3 roll       ( xu...x2 selector1 flag1 selector1 n-top )
   =  or        ( xu...x2 selector1 flag2 )
;
```

Figure Seven-b.

```
: (set-of)  ( xu...x1 -- )  ( r: selector depthu return-addr -- )
   r>              ( xu...x1 my-ret )  ( r: selector depthu -- )
   depth r>  -     ( xu...x1 my-ret  number-of-items-to-check )  (
r:selector )
   swap  r>   swap >r  ( xu...x1 #-of-items-to-check selector ) (
r:"restored")
   false rot           ( xu...x1 selector false number-items )
   0 do  in-set  loop
;

: ] set-of  ( -- addr )  \  terminates a set in a CASE statement
   compile (set-of)  [compile] if   compile drop
; immediate
```

```
ok 110 60 do i sample loop
60 default
61 default
62 default
63 default
64 default
65 default
66 default
67 default
68 default
69 default
did-this-stuff
71 default
72 default
73 default
74 default
75 default
76 default
77 default
78 default
79 default
did-this-stuff
81 default
82 default
83 default
84 default
85 default
86 default
87 default
88 default
89 default
did-this-stuff
91 default
92 default
93 default
94 default
95 default
96 default
97 default
98 default
did-this-stuff
did-some-other-stuff
101 default
102 default
did-quite-a-different-thing
104 default
105 default
106 default
107 default
108 default
109 default
```

The final version you see here was pasted to the interpreter, and the results were pasted back to my editor. An interpreter with a windowing system is a nice combination!

Perhaps this is another case of a programmer with too much time on his hands. However the need was real, and the actual code didn't take long to write. It's not going to have blinding speed but, if the need for speed appears, we will fill it at that time. In the meantime, this is an example of how flexible the Forth control structures can be if you understand how they work.

Randy Leberknight's interest in Forth is natural: for 19 years, he has been working in areas where hardware and software mix. First, he spent 11 years at a company which made and used software for printed circuit board layout—using software to help make hardware. Then he spent about seven years at Forth, Inc., learning Forth, teaching Forth, using Forth, and helping to create new Forths. For the past year, he has been at Motorola Computer Group, in Tempe, Arizona, working on Open Firmware for PowerPC-based systems. He says one of his favorite parts of the job is "...using the interpreter's debugging facilities to help bring up new hardware. It is gratifying to have the engineer who designed the board ask me to show him how I figured out which part was not working!"

Figure Eight.

```
: do-this-stuff
  ." did-this-stuff "  cr ;

: do-some-other-stuff
  ." did-some-other-stuff"  cr ;

: do-quite-a-different-thing
  ." did-quite-a-different-thing"  cr ;

: sample ( selector -- )
   case
     set[  70 80 90 99 ] set-of  do-this-stuff endof

     100 of do-some-other-stuff endof
     103 of do-quite-a-different-thing endof

     dup . ." default "  cr
   endcase
```

Mining the Contents of FD Volume XI

So you have some spare time on your hands, and you've decided to write a multiprocessor-based, multi-tasking, real-time expert system which offers local variables, a set of 8250 UART words, access to 16 Mbytes of PC memory, and linkage to externally assembled programs. Are you nuts? No, you've got a copy of *Forth Dimensions* Volume XI, where these concepts are all discussed and illustrated.

Okay, so you don't need something quite so lofty. You'd just like to write a more useful memory dump routine for your system. Well, you'll also find that and more in Volume XI. What? You don't have Volume XI in your personal library? You're in luck. The FIG office still has a few copies in stock. Here is a brief review of the papers you'll find in Volume XI.

Issue Number 1

Allen Anway presents his "VVDUMP" Extended Byte Dump application. Tired of having to enter an address and count for every region of memory he wants to examine, Allen writes a memory browser which can move forward and backward through memory. He also revises the display of memory, printing the ASCII representation of each byte directly below its hex value, making the dump far more readable. Last, Allen gives us all the rope we need to hang ourselves, by adding the ability to enter values directly into memory. A well-written and useful paper, the only thing I would change is to display the dump with the memory addresses increasing *up* the page rather than down.

This issue contains three papers on local variables: one by Jyrki Yli-Nokari, one by John Hayes, and one by Jose Betancourt. Yli-Nokari is a man after my own heart. Of the three implementations, his is the simplest. It's also the easiest to convert for your own Forth system, since it's built from only *three* blocks of source code! (And one of those is a load block!) Hayes' implementation, based on the concept of *scopes*, has a somewhat more complex syntax, but offers the added feature of *named locals*, where a local variable's name is declared within the definition of its use. Of course, this added capability has a cost, requiring the slight modification of the Forth system's dictionary-search-and-management words. Betancourt's implementation also provides for named locals, but uses a set of prefix operators to find them at compile time. This technique is more portable than Hayes', but makes for a somewhat less readable syntax. If your Forth system doesn't recompile itself from source code, this technique may be the easiest way for you to implement named locals. Interestingly, the three authors offer these comments: "Finally, it seems that local variables are not very useful in everyday work, since we already have the stack for temporary values." (Yli-Nokari); "Many Forth definitions are simple enough that nothing would be gained by using local variables." (Hayes); "Local variables may not be required if definitions are kept short and the stack is kept shallow." (Betancourt). Keep this in mind. Should you find yourself desperately in need of locals, you've probably solved your problem wrong. Of course, there are always those days when you just can't seem to get the problem properly decomposed, and on those days, until you can return and rethink the problem, locals can come in handy.

In his paper, Ayman Abu Mostafa asserts that Forth needs three more stacks. Dissatisfied with the traditional Forth use of the return stack for temporary and index storage, he provides a separate *auxiliary* stack for this use. Next, to provide for the interpretive use of `IF ... ELSE ... THEN` constructs, he creates the *condition* stack. Fortunately for us, our now-standard `[ IF] ... [ ELSE] ... [ THEN]` system has proved to be *far* simpler to implement and use than the modifications required by his system. Last, he provides a *case* stack for the interpretive use of `CASE` statements. While his paper is interesting from an academic standpoint, his premise that using the return stack for "...storing indexes and limits of `DO` loops, and for temporary storage... is bad programming" has, in my experience, yet to be proven in a production environment.

Last, Brian Fox found the limits of his Forth system when he needed to write a high-speed, serial interface to control a video tape recorder. A true Forther, he dauntlessly develops a beautifully decomposed lexicon to control the PC's 8250 UART. Brian is another man after my heart, saying things like, "...it occurred to me that the English words we use... are perfect Forth syntax." Emphasizing the style and techniques that give Forth its real power, and providing a widely needed utility, this nicely written paper is a must-read for the novice and experienced Forther alike.

Issue Number 2

Robert Garian presents a paper and wordset for drawing ovals on the PC's monitor in graphics mode. Robert states, "Ovals have a certain aesthetic appeal that I find a relief from all of the straight lines and rectangles we usually see on computer screens."

Do you like your ovals filled? Zbigniew Szkaradnik presents a paper describing two filling algorithms. The first is designed to fill a closed outline which has already been displayed on the monitor. The second is designed to draw and fill a polygon defined by a list of vertex coordinates. Zbigniew's descriptions of the algorithms are straightforward, and the included source code is surprisingly concise. His use of nested loops will keep you on your toes for at least a few minutes, should you need to reverse-engineer his code.

Frans Van Duinen presents his PDE (Program Development Environment) editor. A block editor, this system is chock-full of features: editing any number of files concurrently, copying and pasting between files, a single-stepper taking its input directly from the editing screen (point-and-step, you might say), from-the-screen `SEE` (decompiler) and `VIEW` (source code locator) functions, source code loading from the

Rich Wagner • Manitou Springs, Colorado
drsavage@usa.net

screen, and a scrolling debug window. Written to load over F83, this editor may require a bit of modification to run within your Forth system, but it may be well worth the effort.

Still working within the confines of a PC-based, 16-bit Forth? Richard F. Olivo provides a set of words for accessing up to 16 Mb of the PC's extended memory through BIOS interrupt 15h. Interestingly, he wrote these words while working with a DT frame grabber. I went through this very same process about six months later, when I too was learning to use the very same frame grabber. Had I been a FIG member when this issue came out, I could have saved myself some time and effort. FIG membership really *can* pay off! Richard's explanatory text is very well written and easy to follow. The source code occupies (get this) only two blocks! That's a huge payoff for so little code.

Marcos Cruz offers his SISIFOrth expert system toolkit. This is a set of words you can use to develop your very own expert system. The code is fairly straightforward, and appears to be pretty portable. Some of Marcos' code is written with a very assembler-like style, so if you, like me, are a very traditional (Chuck suggests the term, *classic*) Forther, you may want to keep a bottle of aspirin handy. Marcos also includes an example expert system, clearly illustrating the use of his toolkit.

This is less than mystical, being of surprising length.

Shades of my old Fortran days, but in reverse! Darryl C. Olivier shows us how to link to an externally assembled program—by enveloping it within our Forth memory image! Noting that a large machine code routine or subsystem can sometimes be written more easily using a full-blown macro assembler than by using the assembler in our Forth systems, Darryl shows us how to pull an externally assembled program into our Forth executable image, jump to it at run time, and return back to our Forth system. His well-written text deftly explains what is otherwise a rather deep subject, and the four blocks (!) of code required to accomplish this can be understood by any more experienced Forther.

Issue Number 3

Dave Edwards provides a set of powerful timing words which can be used in time-critical applications. These words provide a lexicon making it easy to make, say, a routine that runs every 20 milliseconds, or perhaps another that measures its own elapsed execution time or that times an external event. Dave must have decomposed this problem well, because the basic words can be used to construct all kinds of higher-level timing words. The source code is only about 3 Kbytes long, and should port to nearly any system. Dave also provides a plethora of examples showing these words in use.

Examples of engineering and scientific applications of Forth are appallingly few and far between, so I was overjoyed to see Antonio Lara-Feria's and Joan Verdaguer-Codina's paper on a Quaternion Rotation Calculation. Unfortunately, the authors have offered no explanation of the quaternion technique itself (what a bummer, that would have been fun!), and also say little about the accompanying source code. While the code is written in a straightforward fashion, it's not trivial (17 blocks), and may require a math text for its reverse engineering and application.

I've enjoyed Brad Rodriguez' papers ever since I saw him present at my first Rochester Forth Conference (1990). Perhaps it's just the projects he gets to work on, but Brad's papers always seem to tackle deep subjects that come in very handy. His Multiprocessor Forth Kernel in this issue is no exception. In it, he describes a Forth kernel that doesn't just multi-task, but can do so across an arbitrary number of processors. There are a lot of great, new ideas on kernel, operating system, and real-time system architecture in this paper. As always, Brad's writing is clear, concise, and well thought out. This paper is a must-read for anyone writing his own Forth system. I'm sure I'll be stealing ideas from it for some time.

Chester Page presents an architecture and small lexicon for setting vocabulary search orders. The new lexicon eliminates `ONLY` and `ALSO`, and replaces them with two new words. Chester doesn't offer a comparison between his new lexicon and other techniques, so it's hard to say whether or not he's actually got something here. Find out for yourself. The small (about 2 Kb) source listing is easy to understand and re-implement.

Issue Number 4

Nathaniel Grossman presents his Fibonacci Random Number Generator (FRNG). There are a number of techniques for generating pseudo-random numbers. This one ranks among the more complex, both in theory and implementation. Making the complexity seem worthwhile, Nathaniel points out that, by combining a FRNG with a linear, congruential generator, we can build a generator which passes all known tests for randomness. Unfortunately, he doesn't show us how to actually combine the two. While Nathaniel walks us step-by-step through the process of realizing the generator in code, his very mathematician-like writing style can make the going slow at times.

J.B. Ho, P.Y. Kokate, M. Huda, R. Haskell, and N.K. Loh present an application of Forth in Optimal Control. In the paper, they show how a linear quadratic regulator (LQR) can be written in Forth, and applied to the control of a statically unstable system (the old ball-on-a-moving-hill problem). Unfortunately, their explanation of the theory behind the LQR assumes that the reader has a background in Control Theory. Their accompanying source code is a bit cryptic, with word names like `TSTH.CCF`. Luckily, there are only 12 blocks of code (six for a PC implementation, and six for a 68HC11 implementation), making the necessary study and reverse-engineering of the system feasible.

Of historical interest, Howard Rogers shows how to increase the useful Forth dictionary space of a TI 99/4A computer by moving array and heap storage out into the RAM of the video display processor. What's that? You're still using one of these machines? Can I interest you in a slightly used Data General Nova? Anyway, the text and source code are both easy to read, so have fun.

Mike Elola, like many of us who have written production software, is stuck by Forth's lack (actually, *all* programming languages suffer at the foot of this problem) of numeric input routines. Mike's solution to the problem is good, providing a small set of words that will satisfy most of our needs. What's really special about this paper, though, is that Mike takes us along on his quest to find The Simple Solution, as he tries a number of tacks in an effort to identify The Real Problem. The numeric input problem is a tough one to surmount, and Mike's well-written narrative of his trip through solution-land is highly educational. It's a vitally important facet of the programming process which more of us need to include in our papers. Well done, Mike.

Issue Number 5

J.J. Martens provides us with a Double Entry Bookkeeping application. While he doesn't get deeply into the actual mechanics of bookkeeping, he does provide a short description of his system and its embodying source code. His coding style is traditional and straightforward, and the system occupies a total of 15 blocks (plus 15 shadow blocks of documentation). While that size makes the system nontrivial, the code is decomposed very nicely, and is easy to understand. If I were to meet J.J., I'm sure I'd like him. He says things like, "Screen 18 is my favorite... It may not look like much, but it may be where I learned the meaning of iteration. Early versions used up to three screens." Yep, that's Forth. It stays out of your way, so you can concentrate on the process of actually solving your problems.

It's a vitally important facet of the programming process which more of us need to include in our papers.

Chester H. Page presents three evolutionary versions of his Step Trace. The code is written largely in an assembler with which I am not familiar, so it's difficult (without spending a lot of time) for me to say just how easy this code will be to port. That said, each of the three versions of Chester's stepper occupies about four blocks, so there aren't volumes of code to reverse engineer. I have found single-steppers to come in handy on occasion (I've written two in production development environments), so you may want to look into this paper yourself.

Tim Hendtlass is another one of my favorite authors. His paper on Multitasking and Controlling Regular Events is, for the most part, a very well-written tutorial on the Forth cooperative multitasker. Using his paper, any novice would be able to write a task, link it into the chain, start it, and stop it. Tim also includes a short tutorial on defining words, clearly explaining how the `CREATE ... DOES>` construct works. Interestingly, this paper's most significant contribution occurs in the first few paragraphs, where Tim offers an architecture and tiny little lexicon for making task execution time-dependent. For me, this is yet another paper from which I will be stealing ideas.

Do you work with databases? Then David Arnold's Binary Table Search could come in handy. By doing a recursive bisection of a numerically fielded, ordered table, David's binary search can provide an order-of-magnitude performance increase over a typical, sequential search. If your system has a lot of tables in it, then including the binary search as a low-level tool could give your system quite a performance boost. David's explanation of the search technique, and of his code, is well written and very clear. The business end of his code occupies only four blocks, making it easy to port to your system. David also provides an example application for even more clarity.

Last, Jack Woehr waxes eloquently about Forth and its realization in hardware in his paper, Seeing Forth. The third chapter of his book by the same name, the short text leaves you feeling as if Forth may be something special, perhaps even mystical. (Those who know me will tell you that I disagree.) Jack also includes the source code listing for his SC32 assembler. This is less than mystical, being of surprising length (I'd say about 20 Kbytes of source code). The code itself appears simple enough, however, and may not be very difficult to understand, should you decide to delve into it.

Issue Number 6

Thinking about designing your own Forth processor? Then you won't want to miss this issue.

Phil Koopman presents a wonderful paper offering a historical, as well as technical, perspective of his tenure as a Forth processor designer. In the paper, he describes the process he went through from the time he began his first design, the WISC 16 (for Glen Haydon's WISC Technologies), through his development of the Harris RTX 32P (which, I believe, eventually became the RTX 4000). He also discusses some of the important design decisions and tradeoffs which need to be considered during the Forth hardware design process. This paper offers something very special—a historical perspective on how a technical feat was accomplished. It's a must-read for the Forth programmer and otherwise-technically-trained person alike.

John Hayes describes the design of the SC32, one of the many Forth processors which he and his team designed at the Johns Hopkins Applied Physics Laboratory. While John does a great job of explaining many of the internals of the SC32, he also provides a deft discussion of the reasoning behind, and implementation techniques used in, native Forth processors. John's discussion of the SC32 is so clear and simple that it could be used as a primer on processors for the novice programmer. After reading the paper, I was impressed by the team's insights, particularly their multiple-reuse of certain instructions, providing a number of different Forth operations from a *single* (!) instruction. This is a very interesting paper on a very interesting processor.

C.H. Ting presents his Phase ANgle Difference Analyzer (PANDA). Ting shows us how a very simple algorithm can be used to determine the phase difference of a signal received by two separate sensors. The phase difference can then be converted into an angle of arrival, indicating the direction to the source of the signal. Ting then shows how he implemented this system on a Novix NC4000, providing an accuracy of 0.05 degrees over a frequency range from 20 Hz to 20 kHz. Ting's source code for the system (written in cmForth, still my favorite Forth system) is about 10 Kbytes long and, as always, is clear, well decomposed, and well thought-out. Ting's discussion of the difference algorithm does a good job of explaining a fairly complex subject. However, you'll still have to exercise the old noggin a bit to keep up.

Wondering how to use some of the new ANS Forth words? John Hayes shows how some of the new words can be used to write code which is portable across machines of differing data and address sizes. I expect this paper will be particularly enlightening to those who have only worked on PCs, or to those who have looked at the ANS Forth memory access words asking, "Why?" Here is why, and how. Note that he wrote this paper long before the standard was cast in stone. I didn't note any significant differences between the words as he used them and as they appear today but, as always, your mileage may vary.

Epilogue

"There, I am done. Whew, that was hard work."
—Ed Norton
The Honeymooners

Writing this review required about eight hours of work per issue, making for a total of about 48 hours. Why did I do it? At first, it was because Skip and Trace Carter are my friends, and they asked me to do it. However, shortly after starting the project, I began to realize that I was getting something out of it myself—an exposure to all this great work others had done. Lately, with my limited free time, I rarely get to read my *Forth Dimensions* cover-to-cover. For the review, however, I read each issue carefully and thoroughly, learning a lot of great things in the process. On top of that, I really feel I've provided a service to the Forth community, and I'm pretty happy with the results.

I guess my point is this: don't be afraid to donate your time and energy to FIG. In the end, you'll find it pays off as much for you as it does for others.

Rich Wagner, a "starving aerospace engineer," has been writing production software for more than ten years. Having written "more than his share" of FORTRAN, C, and Ada, he began using Forth while developing the Sensor Driven Airborne Replanner—an autonomous, robotic aircraft control system—for the U.S. Navy. He's since written Forth software for MCI, Digalog Corp., and Technology Associates of Colorado, and is now a Senior Engineer at iTV Corporation, helping to develop the Pegasus TV set-top Internet computer.

"Letters," continued from page 5

patching mechanism was not original with Bob, I don't know where he got it, but he was using it by 1983.

2. My point was not to propose something new, but rather to re-propose something old, of clear usefulness, and familiar enough that we could all accept it.
3. I believe I have a healthy regard for the ANSI Forth Standard. My feeling is that, if something can be done in a standard way, it should be; but if I need something the standard doesn't provide, I'm not going to deprive myself of it on purist grounds. My word `RETRY`, which is central to my coding style, is a case in point.
4. I agree that the technique of dictionary patching with which I implement the `MODULE` idea is not portable, though I'm assured that the idea can also be implemented with hashed headers. In any case, it probably cannot be implemented in a standard way, at least not with the design goals I had in mind. These design goals are:
   i. To keep the implementation part of a module private and localized. In particular, I wanted to prevent the user (me) from opening the private part later and adding to the private part. This has to do with keeping things clean.
   ii. To not only hide the headers, but also to throw them away and reclaim the space.
   iii. To make these modules nestable.

   Wil's implementation of `GENERAL` and `PRIVATE`, if I understand it, while interesting, meets none of these goals and would have been of no use to me in the project where I needed my implementation. Which brings me to my last point:
5. I often wonder, when I see neat new Forth ideas, whether they've been implemented just because the implementor could, or whether they meet an actual need in an actual product. When I was first exposed to Mitch Bradley's implementation of LISP's `CATCH` and `THROW` by Mike Perry at the tail end of a FORML session several years ago, I adopted it immediately. I may have been among the top ten users of these words in the Forth world before ANS Forth, because they met a need for me (not error handling, by the way, but menu navigation). (All right, `CATCH` and `THROW` are cute, too, and I liked that.) `RETRY` and my `MODULE` implementation have also received extensive use—hundreds, thousands of uses in the unfortunately now-defunct food service application I used to tend... I had two versions of my `MODULE` implementation words (the second being the one that uses `EXPORTS`) because the first, which just patches the dictionary, didn't meet the needs of my actual practice. On the other hand, I've never had any use for local variables, and thus have not spent any energy developing clever implementations of them, standard or otherwise.

Finally, I apologize to Val Shorre for not knowing his work, and I apologize to all to whom I may have seemed to be taking credit for something I didn't mean to take credit for. Again, my intent was to try to get a useful idea used, not to stake out territory.

Richard Astle • Del Dios, California
rastle@bigfoot.com
rastle@ix.netcom.com

Working Comments (long)?

Abstract—This note describes the genesis of a (possibly) useful Forth tool. Code fragments can be tested prior to compilation to determine their effect on the data and return stacks, without risking system crashes or hidden bugs. Code for a preliminary version is given, together with discussion of possible improvements, should that prove desirable.

The newsgroup comp.lang.forth is, as everyone knows, a wonderful sector of cyberspace. Not only is it inhabited by some of the most helpful and well-mannered folks one is likely to encounter anywhere, it is often a fertile source of ideas.

The title of this article was the subject heading of a conversation that took place in comp.lang.forth late last summer (1996). M. Jean-François Brouillet, a relative newcomer to the newsgroup, inquired whether anyone else had been troubled by the problem of manipulating deep stacks (and losing track of them). I quote (with permission) excerpts from M. Brouillet's post:

Subject: Working Comments (long)?
From: verec@micronet.fr (Jean-Francois Brouillet)
Date: 1996/08/31 Newsgroups: comp.lang.forth

[introductory remarks deleted]

a) the situation: I find myself too often lost with an untractable series of stack operations in a row, and, even if I put stack comments line by line, I have to make sure they reflect correctly what is happening on the stack. This means I have to do the check twice: one for the correct sequence of DUPs, OVERs, NIPs, etc.... the other to make sure the stack comment really reflects what's going on.

So it occurred to me that, since keeping track of the stack state required that stack comments be written, why not then write only stack comments, and make them work instead of the corresponding sequence of stack operators?†

b) the goal: make (a b c d -- b d) actually work.

c) limitation: nothing but stack items already on the stack can be expected to be on the stack after completion. In other words, I don't want to include a full language within the permutation operation.

d) extension: handle the return stack, too.

[syntactic notations deleted]
[potential implementation strategies deleted]

Comments, anyone ?
Jean-Francois Brouillet, verec@micronet.fr
Macintosh Software Developer

†Brouillet's complaint reminds me of how I came to write a FORmula TRANslator [1] and a finite state machine compiler [2]. I was appending formulaic comments to programs that evaluated mathematical expressions, and state transition table comments to programs that used the finite state machine style of programming. At some point in each case, it belatedly occurred to me the computer could do much of the work, by compiling such comments into Forth code.

Julian V. Noble • Charlottesville, Virginia
jvn@virginia.edu

My reply:

Subject: Re: Working Comments (long)?
From: jvn@faraday.clas.Virginia.EDU (Julian V. Noble)
Date: 1996/09/02 Newsgroups: comp.lang.forth

verec@micronet.fr writes:
[deleted]
> d) extension : handle the return stack too.
[more deleted]
> Comments, anyone ?

You would certainly have to include the return stack, since the stack picture

```
        ( a b c d -- b d)
```

is ambiguous. That is, there are several ways to get this result, each with different program consequences:

```
        :2nip  nip rot drop ; ( a b c d -- b d)
        : shuffle_off_to_buffalo  swap >r rot >r ;
        : something_else  YOUR CODE GOES HERE ;
```

I frankly don't think the stack picture compiler is very useful.

However, you might want to consider a stack-picture producer—that might be a useful interactive tool. That is, confronted with something like:

```
        SP" tuck >r  4 roll  swap"
```

it might respond with

```
        tuck >r 4 roll swap
        assumes 5 arguments
        leaves 1 argument on return stack
        ( n4 n3 n2 n1 n0 -- n3 n2 n0 n1 n4) ( r: -- n0)
```

Julian V. Noble, jvn@virginia.edu

(Before anyone hastens to point out that the word `shuffle_off_to_buffalo` is a system-crasher, let me assure everyone I knew this when I posted it.)

Were I to re-post my reply today, I would add that a stack-picture compiler would require much more time to implement than I initially surmised, as it is impossible. ("The difficult takes us a while; the impossible a little longer."—motto of the Seabees.) The reason is implicit in my first answer: the mapping from code to stack picture is many-to-one, hence there is no unique way—short of telepathy or clairvoyance—to reconstruct the intended code. On the other hand, simplifying programs and eliminating bugs are always worthwhile goals, hence my counter-proposal for a code tester—a tool that would reveal the stack effect of a code sequence without crashing the system.

The main question in my mind was whether such a tool was useful enough to justify the effort of its creation. Forth practitioners already know how to simplify their code: factor, factor, factor. By breaking programs into short, telegraphically named subroutines that do simple, definite things, one tends to avoid trivial coding errors arising from stack juggling, especially if one tests new definitions as they are conceived.

FORTH INTEREST GROUP MAIL ORDER FORM

HOW TO ORDER: Complete form on back page and send with payment to the Forth Interest Group. All items have one price. Enter price on order form and calculate shipping & handling based on location and total.

FORTH DIMENSIONS BACK VOLUMES

A volume consists of the six issues from the volume year (May–April).

Volume 1 *Forth Dimensions* (1979–80) 101 – $35

Introduction to FIG, threaded code, TO variables, fig-Forth.

Volume 6 *Forth Dimensions* (1984–85) 106 – $35

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 – $35

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 – $35

Interrupt-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 9 *Forth Dimensions* (1987–88) 109 – $35

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 10 *Forth Dimensions* (1988–89) 110 – $35

dBase file access, string handling, local variables, data structures, object-oriented Forth, linear automata, stand-alone applications, 8250 drivers, serial data compression.

Volume 11 *Forth Dimensions* (1989–90) 111 – $35

Local variables, graphic filling 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 – $35

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, metacompilation.

Volume 13 *Forth Dimensions* (1991–92) 113 – $35

Volume 14 *Forth Dimensions* (1992–93) 114 – $35

Volume 15 *Forth Dimensions* (1993–94) 115 – $35

Volume 16 *Forth Dimensions* (1994–95) 116 – $35

Volume 17 *Forth Dimensions* (1995–96) 117 – $35

FORML CONFERENCE PROCEEDINGS

FORML (Forth Modification Laboratory) is an educational forum for sharing and discussing new or unproven proposals intended to benefit Forth, and is an educational forum for discussion of the technical aspects of applications in Forth. Proceedings are a compilation of the papers and abstracts presented at the annual conference. FORML is part of the Forth Interest Group.

1981 FORML PROCEEDINGS 311 – $45
CODE-less Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectored data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic cueing language for multimedia, NEXOS—a ROM-based multitasking operating system. *655 pp.*

1982 FORML PROCEEDINGS 312 – $30
Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE looping words, number-input wordset, I/O vectoring, recursive data structures, programmable-logic compiler. *295 pp.*

1983 FORML PROCEEDINGS 313 – $30
Non-Von Neuman 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 pp.*

1984 FORML PROCEEDINGS 314 – $30
Forth expert systems, consequent-reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, HP71B Forth, NEON—object-oriented programming, decompiler design, arrays and stack variables. *378 pp.*

1986 FORML PROCEEDINGS 316 – $30
Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple-inheritance system, automatic programming environment. *323 pp.*

1988 FORML PROCEEDINGS 318 – $40
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 pp.*

1989 FORML PROCEEDINGS 319 – $40
Includes papers from '89 euroFORML. Pascal to Forth, extensible optimizer for compiling, 3D measurement with object-oriented Forth, CRC polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RTX recompiler for on-line maintenance, modules, trainable neural nets. *433 pp.*

1992 FORML PROCEEDINGS 322 – $40
Object-oriented Forth based on classes rather than prototypes, color vision sizing processor, virtual file systems, transparent target development, Signal processing pattern classification, optimization in low-level Forth, local variables, embedded Forth, auto display of digital images, graphics package for F-PC, B-tree in Forth *200 pp.*

1993 FORML PROCEEDINGS 323 – $45
Includes papers from '92 euroForth and '93 euroForth Conferences. Forth in 32-Bit protected mode, HDTV format converter, graphing functions, MIPS eForth, umbilical compilation, portable Forth engine, formal specifications of Forth, writing better Forth, Holon – A new way of Forth, FOSM, a Forth string matcher, Logo in Forth, programming productivity. *509 pp.*

1994–1995 FORML PROCEEDINGS *(in one volume!)* 325 – $50 NEW

Fast service by fax: 408-373-2845

BOOKS ABOUT FORTH

ALL ABOUT FORTH, 3rd ed., June 1990, Glen B. Haydon 201 – $90

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. *504 pp.*

eFORTH IMPLEMENTATION GUIDE, C.H. Ting 215 – $25

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 pp.* (w/ disk)

Embedded Controller FORTH, 8051, William H. Payne 216 – $76

Describes the implementation of an 8051 version of Forth. More than half of this book contains source listings (w/disks C050) 511 pp.

F83 SOURCE, Henry Laxen & Michael Perry 217 – $20

A complete listing of F83, including source and shadow screens. Includes introduction on getting started. *208 pp.*

THE FIRST COURSE, C.H. Ting 223 – $25

This tutorial's goal is to expose you to the very minimum set of Forth instructions you need 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 pp.*

THE FORTH COURSE, Richard E. Haskell 225 – $25

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 pp.* (w/disk)

FORTH NOTEBOOK, Dr. C.H. Ting 232 – $25

Good examples and applications. Great learning aid. polyFORTH is the dialect used. Some conversion advice is included. Code is well documented. 286 pp.

FORTH NOTEBOOK II, Dr. C.H. Ting 232a – $25

Collection of research papers on various topics, such as image processing, parallel processing, and miscellaneous applications. *237 pp.*

F-PC USERS MANUAL (2nd ed., V3.5) 350 – $20

Users manual to the public-domain Forth system optimized for IBM PC/XT/AT computers. A fat, fast system with many tools. *143 pp.*

F-PC TECHNICAL REFERENCE MANUAL 351 – $30

A must if you need to know the inner workings of F-PC. *269 pp.*

INSIDE F-83, Dr. C.H. Ting 235 – $25

Invaluable for those using F-83. *226 pp.*

OBJECT-ORIENTED FORTH, Dick Pountain 242 – $37

Implementation of data structures. First book to make object-oriented programming available to users of even very small home computers. *118 pp.*

STARTING FORTH (2nd ed.), Leo Brodie 245 – $37

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 pp.*

THINKING FORTH, Leo Brodie 255 – $30

BACK BY POPULAR DEMAND! The bestselling author of *Starting Forth* is back again with the first guide to using Forth to program applications. This book captures the philosophy of the language to show users how to write more readable, better maintainable applications. Both beginning and experienced programmers will gain a better understanding and mastery of such topics: Forth style and conventions, decomposition, factoring, handling data, simplifying control structures. And, to give you an idea of how these concepts can be applied, *Thinking Forth* contains revealing interviews with real-life users and with Forth's creator Charles H. Moore. To program intelligently, you must first think intelligently, and that's where *Thinking Forth* comes in. Reprint of original, *272pp.*

WRITE YOUR OWN PROGRAMMING LANGUAGE USING C++, Norman Smith 270 – $16

This book is about 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 pp.*

WRITING FCODE PROGRAMS 252 – $52

This manual is written for designers of SBus interface cards and other devices that use the FCode interface language. It assumes familiarity with SBus card design requirements and Forth programming. The material covered discusses SBus development for both OpenBoot 1.0 and 2.0 systems. *414 pp.*

"We're Sure You Wanted To Know..."

Forth Dimensions, Article Reference **151 – $4**
An index of Forth articles, by keyword, from *Forth Dimensions* Volumes 1–15 (1978–94).

FORML, Article Reference **152 – $4**
An index of Forth articles by keyword, author, and date from the FORML Conference Proceedings (1980–92).

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.*

FLOAT4th.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 — C002 – $6
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**

The Math Box, Nathaniel Grossman — C006 – $10
Routines by foremost math author in Forth. Extended double-precision arithmetic, complete 32-bit fixed-point math, & auto-ranging text. Incl. graphics. Utilities for rapid polynomial evaluation, continued fractions & Monte Carlo factorization. Incl. source & docs.
★★ **IBM, 118 Kb**

AstroForth & AstroOKO Demos, I.R. Agumirsian — 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, 700 Kb**

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

8051 Embedded Forth, William Payne — C050 – $20
8051 ROMmable Forth operating system. 8086-to-8051 target compiler. Incl. source. Docs are in the book *Embedded Controller Forth for the 8051 Family.* Included with item #216
★★★ **IBM HD, 4.3 Mb**

68HC11 Collection — C060 – $16
Collection of Forths, tools and floating-point routines for the 68HC11 controller.
★★★ **IBM HD, 2.5 Mb**

F83 V2.01, Mike Perry & 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). Base for other F83 applications.
★ **IBM, 83, 490 Kb**

F-PC V3.6 & **TCOM** 2.5, Tom Zimmer — C200 – $30
A full Forth system with pull-down menus, sequential files, editor, forward assembler, metacompiler, floating point. Complete source and help files. Manual for V3.5 available separately (items 350 & 351). Base for other F-PC applications.
★ **IBM HD, 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 HD, F-PC, 790 Kb**

VP-Planner Float for F-PC, V1.01 Jack Brown — C202 – $8
Software floating-point engine behind the VP-Planner spreadsheet. 80-bit (temporary-real) routines with transcendental functions, number I/O support, vectors to support numeric co-processor overlay & user NAN checking.
★★ **IBM, F-PC, 350 Kb**

F-PC Graphics V4.6, Mark Smiley — C203 – $10
The latest versions of new graphics routines, including CGA, EGA, and VGA support, with numerous improvements over earlier versions created or supported by Mark Smiley.
★★ **IBM HD, F-PC, 605 Kb**

PocketForth V6.4, Chris Heilman — C300 – $12
Smallest complete Forth for the Mac. Access to all Mac functions, events, files, graphics, floating point, macros, create standalone applications and DAs. Based on fig & *Starting Forth.* Incl. source and manual.
★ **MAC, 640 Kb**, System 7.01 Compatible.

Kevo V0.9b6, Antero Taivalsaari — C360 – $10
Complete Forth-like object Forth for the Mac. Object-Prototype access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Kernel source included, extensive demo files, manual.
★★★ **MAC, 650 Kb**, System 7.01 Compatible.

Yerkes Forth V3.67 — C350 – $20
Complete object-oriented Forth for the Mac. Object access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Incl. source, tutorial, assembler & manual.
★★ **MAC, 2.4Mb**, System 7.1 Compatible.

Pygmy V1.4, Frank Sergeant — C500 – $20
A lean, fast Forth with full source code. Incl. full-screen editor, assembler and metacompiler. Up to 15 files open at a time.
★★ **IBM, 320 Kb**

KForth, Guy Kelly — C600 – $20
A full Forth system with windows, mouse, drawing and modem packages. Incl. source & docs.
★★ **IBM, 83, 2.5 Mb**

Mops V2.6, Michael Hore — C710 – $20
Close cousin to Yerkes and Neon. Very fast, compiles subroutine-threaded & native code. Object oriented. Uses F-P co-processor if present. Full access to Mac toolbox & system. Supports System 7 (e.g., AppleEvents). Incl. assembler, manual & source.
★★ **MAC, 3 Mb**, System 7.1 Compatible

BBL & Abundance, Roedy Green — C800 – $30
BBL public-domain, 32-bit Forth with extensive support of DOS, meticulously optimized for execution speed. Abundance is a public-domain database language written in BBL. Incl. source & docs.
★★★ **IBM HD, 13.8 Mb, hard disk required**

Version-Replacement Policy

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

★ – Starting ★★ – Intermediate ★★★ – Advanced

MORE ON FORTH ENGINES

Volume 10 (January 1989) 810 – $15
RTX reprints from 1988 Rochester Forth conference, object-oriented cmForth, lesser Forth engines. *87 pp.*

Volume 11 (July 1989) 811 – $15
RTX supplement to *Footsteps in an Empty Valley*, SC32, 32-bit Forth engine, RTX interrupts utility. *93 pp.*

Volume 12 (April 1990) 812 – $15
ShBoom Chip architecture and instructions, neural computing module NCM3232, pigForth, binary radix sort on 80286, 68010, and RTX2000. *87 pp.*

Volume 13 (October 1990) 813 – $15
PALs of the RTX2000 Mini-BEE, EBForth, AZForth, RTX-2101, 8086 eForth, 8051 eForth. *107 pp.*

Volume 14 814 – $15
RTX Pocket-Scope, eForth for muP20, ShBoom, eForth for CP/M & Z80, XMODEM for eForth. *116 pp.*

Volume 15 815 – $15
Moore: new CAD system for chip design, a portrait of the P20; Rible: QS1 Forth processor, QS2, RISCing it all; P20 eForth software simulator/debugger. *94 pp.*

Volume 16 816 – $15
OK-CAD System, MuP20, eForth system words, 386 eForth, 80386 protected mode operation, FRP 1600 – 16-Bit real time processor. *104 pp.*

Volume 17 817 – $15
P21 chip and specifications; Pic17C42; eForth for 68HC11, 8051, Transputer *128 pp.*

Volume 18 818 – $20
MuP21 – programming, demos, eForth *114 pp.*

Volume 19 819 – $20
More MuP21 – programming, demos, eForth *135 pp.*

Volume 20 820 – $20
More MuP21 – programming, demos, F95, Forth Specific Language Microprocessor Patent 5,070,451 *126 pp.*

Volume 21
MuP21 Kit; My Troubles with This Darn 82C51; CT100 Lab Board; Born to Be Free; Laws of Computing; Traffic Controller and Zen of State Machines; ShBoom Microprocessor; Programmable Fieldbus Controller IX1; Logic Design of a 16-Bit Microprocessor P16 *98 pp.*

MISCELLANEOUS

T-shirt, "May the Forth Be With You" 601 – $18
(Specify size: Small, Medium, Large, X-Large on order form) white design on a dark blue shirt or green design on tan shirt.

BIBLIOGRAPHY OF FORTH REFERENCES 340 – $18
(3rd ed., January 1987)
Over 1900 references to Forth articles throughout computer literature. *104 pp.*

Last 5

DR. DOBB'S JOURNAL back issues

Annual Forth issues, including code for various Forth applications.

September 1982, September 1983, Sepember 1984 (3 issues) 425 – $10

FORTH INTEREST GROUP

100 Dolores St., Suite 183 • Carmel, California 93923 • office@forth.org

For credit card orders or customer service:
Phone Orders ***408.37.FORTH***
weekdays *408.373.6784*
9.00 – 1.30 PST ***408.373.2845** (fax)*

Name	
Company	
Street	voice
City	fax
State/Prov. Zip	e-mail
Nation	

Non-Post Office deliveries: include special instructions.	The amount of your sub-total...	...the shipping & handling
Surface U.S. & International	Up to $40.00	$7.50
	$40.01 to $80.00	$10.00
	$80.01 to $150.00	$15.00
	Above $150.00	10% of Total
International Air		40% of Total
Courier Shipments		$15 + courier costs

PRICES MAY CHANGE WITHOUT NOTICE

Item	Title	Quantity	Unit Price	Total

☐ CHECK ENCLOSED *(payable to: Forth Interest Group)*
☐ VISA/MasterCard:

Card Number exp. date

Signature

sub-total	
10% Member Discount Member#	
Sales tax* on sub-total *(California only)*	
Shipping and handling *(see chart above)*	
Membership* in the Forth Interest Group ☐ New ☐ Renewal $45/53/60	
TOTAL	

✖ MEMBERSHIP IN THE FORTH INTEREST GROUP

The Forth Interest Group (FIG) is a worldwide, non-profit, member-supported organization with over 1,000 members and 10 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 $45 per year for U.S.A. & Canada surface; $53 Canada air mail; all other countries $60 per year. This fee includes $39 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.

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.

SHIPPING & HANDLING: All orders calculate shipping & handling based on order dollar value. *Special handling available on request.*

SHIPPING TIME: Books in stock are shipped within seven days of receipt of the order.
SURFACE DELIVERY:
U.S.: 10 days
other: 30–60 days

***CALIFORNIA SALES TAX BY COUNTY:**
7.75%: Del Norte, Fresno, Imperial, Inyo, Madera, Orange, Riverside, Sacramento, Santa Clara, Santa Barbara, San Bernardino, San Diego, and San Joaquin; **8.25%**: Alameda, Contra Costa, Los Angeles San Mateo, San Francisco, San Benito, and Santa Cruz; **7.25%**: other counties.

Fast service by fax: 408-373-2845

And for cases where the subroutine requires many arguments (Brodie, *Thinking Forth*, p. 204), there are always named VARIABLEs, global or local, to simplify the stack. In fact, M. Brouillet subsequently discovered the LOCALS lexicon of ANS Forth, and found it to be exactly the tool he was seeking, so he lost interest in a stack-effect compiler.

The discussion would have ended there, but for encouragement from Wolfgang Allinger:

Subject: Re: Working Comments (long)?
From: All@business.forth-ev.de (Wolfgang Allinger)
Date: 1996/09/03 Newsgroups: comp.lang.forth

On 02 Sep 96, Julian V. Noble) wrote:
[snipp]
>I frankly don't think the stack picture compiler is very useful.

I had the same feeling, so I didn't think on it. But your answer showed clearly, what I felt. THX.

>However, you might want to consider a stack-picture
>producer—that might be a useful interactive tool. That is,
>confronted with something like
> SP" tuck >r 4 roll swap"
>
>it might respond with
> tuck >r 4 roll swap
> assumes 5 arguments
> leaves 1 argument on return stack
> (n4 n3 n2 n1 n0 -- n3 n2 n0 n1 n4) (r: -- n0)

That's very nice. I would like this function, however I think it's a very complicated/big piece.

Bye bye by Wolfgang, all@business.forth-ev.de
FORTHing @ work Cheap Fast Good ... pick any two of them

Herr Allinger's remark, "That's very nice. I would like this function, however I think it's a very complicated/big piece" challenged me to create a preliminary version of a stack-effect tool. It seemed to me the code need not be large, if one used judiciously the Forth compilation mechanism. So one of my motivations was to see how small one could make a working tool that would still be legible, maintainable, and portable. Another was a wish to respond to oft-posted complaints (in comp.lang.forth) of the scarcity of public-domain code examples that illustrate the power and beauty of Forth, as well as good Forth coding practice*—that is, a professor professes: *c'est son métier.*†

Several days after receiving Allinger's challenge (I doubt he meant it thus, but that's how I took it), I therefore posted the following:

Subject: Re: Working Comments (long) ?
From: jvn@faraday.clas.Virginia.EDU (Julian V. Noble)
Date: 1996/09/09 Newsgroups: comp.lang.forth

All@business.forth-ev.de writes:
[deleted]

*Many Forth professionals, who could doubtless provide much better examples than mine, have been frustrated by contractual arrangements from publishing illustrative samples of Forth at its best.

†This paraphrases what Catherine the Great said when accused of being an autocrat.

> That's very nice. I would like this function, however I think it's a
>very complicated/big piece.

Well, not so complicated. Here is a preliminary version. It is less than 2 pages incl. comments. (Don't know what that is in screens, esp. if made illegible in the usual manner :-)
[Code placed in the Appendix, pp. 22–23 —Ed.]
Let me know if you find it useful as is, or if it needs to output a full stack description, as in the first Usage note.
To do this full stack description needs redefinitions of all Forth words that use stack, so when executed by EVALUATE they don't really execute, but just compute what they do to the stack...
Not hard, just more tedious than I wanted to do last nite.

> Bye bye by Wolfgang, all@business.forth-ev.de
> FORTHing @ work Cheap Fast Good ... pick any two of them

Well, it was cheap and fast, anyway...
Julian V. Noble, jvn@virginia.edu

Since I have tried to make the code almost self-explanatory, it does not need a lot of discussion. The word SP" gets a string, up to the trailing " , and EVALUATEs it (i.e., feeds it to the Forth interpreter). To keep anything untoward from happening, we redefine some standard words in their own VOCABULARY† so they can be interpreted by SP" without performing their normal functions. That is, words like DUP, ROT, etc. that manipulate the data stack can be safely allowed to do their normal thing; whereas words like >R, R>, or R@ are manifestly hazardous. If your test application involves . (dot) or EMIT it is probably a good idea to redefine these, also.

Using the Forth interpreter and eschewing bells and whistles allowed me to keep the code small. The only significant words required (beyond SP" itself) were those defining the simulated return stack and its display. I also included some words that make F-PC compatible with the ANS standard, to facilitate testing by readers without access to a full ANS Forth.

Some final remarks: most good commercial Forths provide single-stepping for debugging purposes. Moreover, keeping words short and factored, testing them as they are defined, eliminates most stack problems. This is why I never felt much need for a stack-picture tool.

I can imagine SP" being useful to experienced Forth programmers in only a few circumstances. One could be having to program in an environment that prohibits the use of tools like single-steppers or decompilers, say, through lack of memory. Another would be to catch errors from unbalanced return-stack manipulations, before they crash a system. Finally, try though we might to avoid such tangles, certain algorithms just seem to demand a stack with more than 1–3 items on it.

References

[1] J.V. Noble, "A FORmula TRANSlator for Forth," *J. Forth App. and Res.* 6 (1990) 131–156.

[2] J.V. Noble, "Avoid Decisions," *Computers in Physics* 5 (1991) 386.

Code begins on next page.

†It is now called a WORDLIST in ANS Forth, for reasons that I cannot fathom—what was wrong with VOCABULARY?

ANS-compatible stack-picture tool

```
\ --preliminary version of 9/9/96
\ --modified for ANS compatibility and bug fixes on 5/24/97
\
\ Author: J.V. Noble   jvn@virginia.edu
\
\     (c) Copyright 1996  Julian V. Noble. Permission is granted
\     by the author to use this software for any application
\     provided the copyright notice is preserved.
\
\ Usage
\     SP" tuck  >r   4 roll  swap"
\     ( r: -- )
\     ( d: 9 8 7 6 5 4 3 2 1 0 )
\     tuck >r 4 roll swap
\     ( r: 0 )
\     ( d: 9 8 7 6 5 3 2 0 4 1 )  ok

\ ANS compatibility (for F-PC)
' 2+  alias  cell+    ' 2-  alias  cell-    ' 2*  alias  cells
: to    state @  if  [compile]  is     else  is     then    ;  immediate

\       -- from eval.seq in the file zimmer.zip
: evaluate       ( a1 n1 --- )
                 dup
                 save!> span     save!> #tib
                 save!> 'tib   0 save!> >in
                 run
                 restore> >in    restore> 'tib
                 restore> #tib   restore> span ;

' vocabulary  alias wordlist
\   --renamed by X3J14 for no reason I can discern!

wordlist   stack-pict                  \ allows redefining >r, etc.
stack-pict definitions                 \ (re)def'ns -> stack-pict

\     CHAR must be (re)defined in the new vocabulary
\     because F-PC uses DEFERed CHAR for something else.

: char  bl word  1+  c@        \ get 1st character of following string
        state @  if   [compile]  literal   then    ;  immediate

\ Code for stack-picture tool begins here

create fake_r  10 cells  allot         \ fake r-stack
fake_r   value  rp                     \ fake r-stack pointer
: r-set   fake_r  to  rp    ;          \ initialize fake r-stack
: >r      rp !   rp cell+ to rp   ;    \ push to fake r-stack
: r@      rp  @  ;
: r>      rp cell-  to rp  r@  ;       \ pop from fake r-stack
: 2>r     >r >r  ;
: 2r>     r> r>  ;

: .r      \ display fake r-stack
          cr  ." ( r: "
          rp  fake_r
          2dup  cell-  <=   abort" rstack underflow! )"
```

```
         2dup  <=   if  ." -- )"  2drop exit then   \ empty r-stack
         do  i @  .  space   1 cells  +loop
         ."  )"   ;

depth   value  old-depth           \ place to save current stack depth

: #items    depth  old-depth  -  ;  ( -- #items)

: .s      \ display data stack -- note difference from usual .s
         cr   ." ( d: "
         #items  ?dup
         if   ( #items)  1-  0 swap  ( -- 0 n-1)
              do   i pick  .  space  -1  +loop   ." )"
         else  ." -- )"   then   \ nothing on stack
         ;

: s-clear    #items  dup
            0>  if   0 do drop  loop    then  ;

: ten_#'s     0 9 do i -1 +loop   ;    ( -- 9 8 7 6 5 4 3 2 1 0)

: initialize    stack-pict             \ set search order
               depth  to  old-depth   \ save present data stack
               r-set                  \ reset fake r-stack
               ten_#'s   ;

forth     forth definitions

: sp"     [  stack-pict ]           \ set search order
         initialize
         .r  .s                    \ display initial stacks
         char "   word             \ get test string
         count   2dup              ( -- c-adr u c-adr u)
         cr  type                  \ display string
         evaluate                  \ evaluate test string
         .r  .s                    \ display final stacks
         s-clear                   \ clean up
         forth   forth definitions  ;
```

Standardizing OOF Extensions

Standardization and Libraries

Andrew McKewan argues that we need to agree on (i.e., standardize) one model to start building an object-oriented library. My view is just the reverse: If someone writes a good object-oriented library that everyone wants to use, we will all use the object model on which that library is based; that model will become a *de facto* standard and, finally, a *de jure* standard.

Andrew McKewan uses the Forth Scientific Library (FSL) as an example. He argues that it was necessary to standardize floating-point math in ANS Forth before the FSL could be written. Even if that were true (of which I am not convinced), the cases differ significantly: a floating-point implementation written in Forth-83 would have been unacceptably slow, for many purposes; on the other hand, many object-oriented models can be implemented in standard Forth efficiently enough to be useful. (By the way, this is a major strength of Forth over many other languages: C, Pascal, Ada, etc. require language changes and new compilers to accommodate object-oriented programming; in Forth, every programmer can do it.)

As a counter-example, consider the case of locals: even though the committee standardized the syntax `LOCALS| this read can you|`, many people use the syntax `{ you can read this}` . This syntax can be implemented without performance penalty in standard Forth[1] (http://www.complang.tuwien.ac.at/forth/anslocal.fs).

We can make the Neon model standard, but should be aware of its properties.

What would such an object-oriented library look like? It should have as few system dependences as possible. This excludes libraries for dealing with windowing systems, which appear to be the most popular application of object-oriented technology in Forth. The library should deal with problems that are hard enough to make reinventing the wheel unattractive. A look at the standard libraries of other object-oriented languages should provide some inspiration.

Consensus

The major problem with standardizing an object-oriented model by agreeing on one is that there is no consensus. There was a discussion of this topic on comp.lang.forth (subject: Objects for ANS Forth) in August 1996. Rodriguez and Poehlman[2] list 17 object-oriented extensions for Forth, and this does not include several that were discussed on comp.lang.forth.

On the other hand, what's so bad about having no standard object model? We don't have a standard array or structure model, either, because we can build what we need when we need it, at little cost.

Anton Ertl • Vienna, Austria
anton@mips.complang.tuwien.ac.at

The Neon Model

I cannot create a synopsis of the complete discussion, therefore I'll restrict myself to the points relevant to the Neon/Yerk model (as presented by McKewan[3]), which is also implemented in Mops, Win32Forth, and in ANS Forth (as presented by McKewan). This model currently appears to be the most popular. The points under discussion were:

- The Neon model uses a *selector object* syntax, which makes it unnatural to pass objects on the stack. This syntax makes it easy to pass the *selector* on the stack, but that is rarely needed.

  The Neon model uses the following syntax for dealing with objects passed on the stack:

  `selector [ code ]`

  `code` must produce an object reference, which is then consumed by the whole construction. This syntax reduces the extensibility (see below), and offers no advantage over the more conventional (and, therefore, easier to learn) syntax

  `code selector`

- The Neon model requires that the selector parse the input stream (at compile time). This leads to reduced extensibility, and to bugs that are hard to find.

  E.g., suppose, for some reason, you want to tick a selector to get an execution token that you can `EXECUTE` or `COMPILE,` later. How do you pass the object to that selector? You cannot use the natural way, which is to pass it on the stack; instead, you have to manipulate the input stream.

  Once you have managed to deal with the input stream, the real trouble starts: All selectors defined with McKewan's implementation of the Neon model are state smart. I.e., what they do depends on the contents of `STATE` when the selector is invoked. So you have to be sure to set `STATE` right for every place where such a selector might be invoked. If you fail in that, the resulting bug usually is hard to find.

  This should demonstrate the trouble with parsing words in general and with the Neon model in particular. We could choose to forbid ticking (and `POSTPONE`ing) selectors, or we could choose a model that does not have this problem.

- The Neon model allows sending (a message with) any selector to any object (let's call such models Smalltalk-like). This is considered essential, by some. However, it also makes it harder to create efficient implementations.

In contrast, some models (let's call them Java-like) allow sending only those selectors to an object that were explicitly defined for the class of the object or its ancestor classes. Some also have *multiple inheritance* or Java-like *interfaces*.

In practice, you can program the same things with the Smalltalk-like and Java-like models. In the Java-like models, you have to define the selector in a common ancestor class (or common interface) of all objects that use the selector. If you fail to do this, and send a message to an object for which the selector was not defined, the result in a straightforward implementation of a Java-like model is a crash or the invocation of an unrelated method; in contrast, with a Smalltalk-like model, you get a run-time error `message not understood`.

Concerning implementation, a Java-like model can be implemented easily and efficiently, using a technique that C++ implementors call *virtual function tables*. For a Smalltalk-like model on an interactive system like Forth, using virtual function tables is much harder.[4] Indeed, as far as I know, no Smalltalk-like Forth extension uses virtual function tables; they all use searching methods that are significantly slower.

Proponents of Smalltalk-like models argue that most selector lookups can be resolved at compile time, eliminating the searching overhead. However, studies of programs written in full object-oriented style (in other languages) show that message sends occur every 60 instructions (median), and even complex analysis algorithms leave a significant number of them unresolved (usually because the message send actually does invoke different methods at run time).[5]

We can make the Neon model standard (after all, we can still implement the others in plain Forth), but if we do so, we should be aware of its properties.

References

1. John R. Hayes. "User-defined local variable syntax with ANS Forth." *SigForth Newsletter*, 4 no. 2, 1992.

2. Bradford J. Rodriguez and W.F.S. Poehlman. "A survey of object-oriented Forths." *SIGPLAN Notices*, pages 39–42, April 1996.

3. Andrew McKewan. "Object-oriented programming in ANS Forth." *Forth Dimensions*, March-April 1997.

4. Jan Vitek and R. Nigel Horspool. "Compact dispatch tables for dynamically typed object oriented languages." In Tibor Gyimóthy, editor, *Compiler Construction* (CC '96), pages 309–325, Linköping, 1996. Springer LNCS 1060.

5. Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. "Simple and effective analysis of statically-typed object-oriented programs." *Conference on Object-Oriented Programming Systems, Languages & Applications* (OOPSLA '96, pages 292–305, 1996.

Anton Ertl has been programming in Forth since 1983. He currently uses Forth as a subject of research focusing on native code compilation (see http://www.complang.tuwien.ac.at/projects/forth.html), and also as a tool for non-Forth research. He is also a coauthor of the threaded code system Gforth, which aims to be both fast and portable.

Continued from page 9

or decreasing the size of a running transputer Forth system).

- Saving a modified transputer Forth system to the current DOS drive via `SAVE-SYSTEM TFORTH.FYS`.
- Saving a modified transputer Forth multisystem to the current DOS drive via `SAVE-MULTI TFORTH.FYM`.
- On-line help for all vocabularies: `HELP <word>`.
- Rich documentation in various .doc files.
- Saving any transputer Forth RAM area as a DOS file.
- Loading the contents of any DOS file to any transputer Forth RAM area.
- Breaking a transputer Forth program any time via `[ ESC]`, thereby jumping to server in host...
- ...resuming such a program via `START`...
- ...or immediately exit to DOS by `BYE-FOR-DOS` or `[ ESC] BYE`.
- Resuming transputer Forth operation from DOS level via `TRESUME` (DOS batch file; data stack contents are preserved).
- New loading of the transputer Forth system via `LOAD-FTP` (the data stack contents get lost).
- New loading of a previously saved multisystem via `LOAD-FTP M` (the data stack contents get lost).
- `DUMP`ing with screen-forward and screen-back option on keystroke.
- `LEARN` setup procedure binds the on-line help system to the respective `CONTEXT` vocabulary; `LEARN` comes with statistics on words documented, words not found in the documentation file, and words defined more than once.
- Keyboard buffer extension facility `KEYBUF128!`.
- String input into keyboard buffer (so parameters to DOS command calls can be easily transferred).
- Loading any transputer Forth program source via `INCLUDE <name.FTH>`.
- 25% of the system gets metacompiled, the remaining part is simply loaded by `INCLUDE TCORE [ RET]`.
- `KERNEL` reduces system to base part of said 25%.
- Easy modification of remaining 75% of transputer Forth system by modifying the source file TCORE.FTH and inputting `KERNEL INCLUDE TCORE`.
- Easy inclusion of non-ASCII characters in word names, by switching from `ENGLISH` to `GERMAN` (during compilation, a length is placed at the end of the word's name field).
- `TASSEMBLER` acts as cross-assembler in host-resident server.
- Assembling and disassembling (e.g., of OCCAM object code) also from server in host.
- `DUMP`ing also from server in host: `TDUMP`.
- Access to any editor via `DOS <name>` (e.g., `DOS EDLIN`).
- Calling an additional host (Turbo-) Forth system from inside F-TP: `DOS FORTH`.
- Calling PCTOOLS from inside F-TP: `DOS PCTOOLS`, etc.
- Improved version of `LIST` allows moving to previous screen.
- Ability to mix high-level and low-level words any number of times while compiling a word.
- 32-bit data and return stacks, addresses, single-precision integers and floating-point numbers (complies with IEEE 754).
- 64-bit double-precision integers and floating-point numbers (in compliance with IEEE 754).

One could even use F-TP 1.00 by simply mouse-clicking on the appropriate icon on the Windows 3.11 or Windows 95 desktop; however, there is no use doing so because, e.g., the list of words invoked by calling `WORDS` will creep extremely slowly over the screen, compared to the enormous speed achieved by operating F-TP 1.00 from DOS.

Get It Up

```
( Six One-Liners )
( All definitions are in Standard Forth CORE and CORE-EXT words. )

: ANEW     >IN @ BL WORD FIND IF EXECUTE ELSE DROP THEN >IN ! MARKER ;
: BOUNDS   OVER + SWAP ;                ( a n -- a+n a )
: HAVING   BL WORD FIND NIP 0= IF POSTPONE \ THEN ; IMMEDIATE
: LACKING  BL WORD FIND NIP IF POSTPONE \ THEN ; IMMEDIATE
: ::       S" ANEW NONCE  : NONCE-DEF "  EVALUATE ; IMMEDIATE
: ;;       S" ; NONCE-DEF  NONCE "  EVALUATE ; IMMEDIATE

0 [ IF] COMMENT

ANEW ( "<spaces>name" -- )

    : ANEW                          ( "<spaces>name" -- )
        >IN @                           ( >in)
            BL WORD FIND IF             ( . exec_token)
                EXECUTE                 ( >in)
            ELSE DROP THEN
        >IN !                           ( )
        MARKER
    ;
```

If *name* is found, ANEW executes it. *name* is expected to have been defined by MARKER. name will forget itself and all definitions after it.

ANEW then executes: MARKER *name*.

This defines *name* again so you can redefine the definitions following it.

Put ANEW *marker-name* at the beginning of your source files.

```
BOUNDS  ( a n -- a+n a )

 : BOUNDS  OVER + SWAP ;  ( a n -- a+n a )
```

Adjust the arguments for a DO-loop.

```
0 [ IF] COMMENT
```

Comment out documentation using [IF] [ELSE] [THEN] .

Used:

```
        0 [ IF] COMMENT
        mumble mumble mumble
        [ THEN]
```

Note that [IF] [ELSE] [THEN] should be kept balanced or, better, not used between [IF] and [THEN] . This restriction applies within comments and quoted strings as well.

My personal convention is to use 0 [IF] or 1 [IF] to select alternative code that should work, FALSE [IF] to comment out code that doesn't work, and 0 [IF] COMMENT to comment out non-code.

I use \ to comment out code temporarily, and for other comments I don't want in the final source. That's why you don't see it much in my code.

I use (for stack state and other comments I want to keep in the final source.

```
HAVING ( "<spaces>name" -- )

   : HAVING            ( "<spaces>name" -- )
       BL WORD FIND NIP 0=
           IF POSTPONE \ THEN
   ; IMMEDIATE
```

If *name* is found, interpret the rest of the line; otherwise, skip the line. This is IMMEDIATE and can be used in a definition.

Wil Baden • Costa Mesa, California
wilbaden@netcom.com

Wil Baden is a professional programmer with an interest in Forth.

To include more than one line when *name* is not found:

```
        HAVING name 0 [ IF]
        _whatever_
        [ THEN]
```

```
LACKING ( "<spaces>name" -- )

    : LACKING           ( "<spaces>name" -- )
        BL WORD FIND NIP
            IF POSTPONE \ THEN
    ; IMMEDIATE
```

If *name* is not found, interpret the rest of the line; otherwise, skip the line. This is IMMEDIATE and can be used in a definition.

LACKING is generally used before defining the name that is lacking.

```
::
```

Start compiling a nonce word. The same as:

```
        ANEW NONCE    : NONCE-DEF
```

Used extensively for data initialization and testing.

Use ;; to complete the definition, execute it, and forget it.

Nonce words let you execute loops from the keyboard. They also allow you to initialize data structures programmatically with no overhead.

Nonce words wouldn't be needed with self-compiling:

```
    IF   BEGIN   DO   ?DO
```

From the *Random House Dictionary:*
nonce word, a word coined and used only for the particular occasion.

```
;;
```

Finish compiling a nonce word, execute it, and forget it. If errors occur, start over or type NONCE to recover. The same as

```
        ;    NONCE-DEF    NONCE
```

Procedamus in pace.
Wil Baden

```
[ THEN] [ THEN] [ THEN]
```

Continued from page 29

```
[ ELSE]

    CREATE upper-case-map   256 CHARS ALLOT

    ( Initialize the map to change characters into themselves. )
    :: 256 0 DO  I  upper-case-map I CHARS +  C!  LOOP ;;

    ( Replace the lower-case letters with upper-case. )
    :: [ CHAR] a  26  BOUNDS DO
            I BL XOR  upper-case-map I CHARS +  C!  LOOP
    ;;

    ( Convert to upper-case by `addr + C@' inline. )
    : UP-CHAR    S" CHARS upper-case-map + C@ " EVALUATE ; IMMEDIATE

    : DOWN-CHAR  BL XOR  UP-CHAR  BL XOR ;

[ THEN] [ THEN]

( Convert string to upper-case. )
: UP-STRING                                    ( a u -- )
    0 ?DO   DUP C@ UP-CHAR  OVER C!  CHAR+   LOOP DROP
;

SEE UP-STRING  CR
```

```
--
Neil Bawd                               Goat Hill, California
"Anywhere is interesting for 15 minutes, except maybe Iowa."
```

THE VIEW FROM GOAT HILL - #01

Speed It Up

Improving String Processing Speed

Before doing anything with this, include *Tool Belt #01.*

String processing in a high-level language, particularly Forth, is notoriously slow. Here are ideas to improve the speed of some Forth definitions.

The example is conversion of a string to upper case.

The first test shows how a competent Forth programmer coded it. For each character in the string, he calls **UP-CHAR,** which calls **BETWEEN,** which calls **WITHIN**. It does five to eight, or more, Forth words besides.

Approximately the following is done for every character.

```
nest DUP lit lit nest 1+ jump OVER - >R - R>
U< unnest BL AND XOR unnest
```

In the second test, five to eight Forth words are compiled in-line, without any intermediate function calls.

The following is done for every character.

```
DUP lit - lit U< BL AND XOR
```

In the third test, two or three Forth words are compiled in-line, at the expense of a 256-character map.

The following is done for every character.

```
addr + C@
```

INCLUDE this file three times to see results. **UP-CHAR** will have a different definition each time. The decompilation of **UP-STRING** will be different each time, although the source is the same.

You can set **Up-Char-Test** to 0 to start over.

ANEW, BOUNDS, LACKING, `::`, and `;;` are defined in *Tool Belt #01.*

Making code in-line by using macros defined with **EVALUATE** can often improve performance dramatically.

The first time the file is INCLUDEd, the result is:

```
VARIABLE Up-Char-Test   0 Up-Char-Test !

ANEW Up-Char-Test-Run

1 Up-Char-Test +!   CR  .( Test# ) Up-Char-Test ?  CR

    : BETWEEN   1+ WITHIN ;                  ( x min max -- flag )

    : UP-CHAR ( c -- C ) DUP [CHAR] a [CHAR] z BETWEEN  BL AND  XOR ;

            SEE WITHIN   SEE BETWEEN   SEE UP-CHAR

( Convert string to upper-case. )
: UP-STRING                                  ( a u -- )
    0 ?DO   DUP C@ UP-CHAR  OVER C!  CHAR+   LOOP DROP
;

SEE UP-STRING  CR
```

The second time the file is included:

```
ANEW Up-Char-Test-Run

1 Up-Char-Test +!   CR  .( Test# ) Up-Char-Test ?  CR

    ( Eliminate nesting  UP-CHAR  BETWEEN  WITHIN  )
    : UP-CHAR                               ( c -- C )
       S" DUP [CHAR] a -  26 U<  BL AND  XOR " EVALUATE
    ; IMMEDIATE

( Convert string to upper-case. )
```

Neil Bawd • Goat Hill, California
NeilBawd@forth.org

Neil Bawd is known by FORML Conference attendees; he should not be mistaken for his evil twin. For the source for this article, send e-mail requesting "The View from Goat Hill: Speed It Up."

```
: UP-STRING                                         ( a u -- )
    0 ?DO   DUP C@ UP-CHAR  OVER C!  CHAR+   LOOP DROP
;

SEE UP-STRING  CR
```

The third time the file is included:

```
ANEW Up-Char-Test-Run

1 Up-Char-Test +!   CR  .( Test# ) Up-Char-Test ?  CR

    CREATE upper-case-map   256 CHARS ALLOT

    ( Initialize the map to change characters into themselves. )
    :: 256 0 DO  I  upper-case-map I CHARS +  C!  LOOP ;;

    ( Replace the lower-case letters with upper-case. )
    :: [CHAR] a  26  BOUNDS DO
           I BL XOR  upper-case-map I CHARS +  C!  LOOP
    ;;

    ( Convert to upper-case by `addr + C@' inline. )
    : UP-CHAR    S" CHARS upper-case-map + C@ " EVALUATE ; IMMEDIATE

    : DOWN-CHAR  BL XOR  UP-CHAR  BL XOR ;

( Convert string to upper-case. )
: UP-STRING                                         ( a u -- )
    0 ?DO   DUP C@ UP-CHAR  OVER C!  CHAR+   LOOP DROP
;

SEE UP-STRING  CR
```

Source Listing

```
LACKING Up-Char-Test   VARIABLE Up-Char-Test   0 Up-Char-Test !
ANEW Up-Char-Test-Run
1 Up-Char-Test +!   CR  .( Test# ) Up-Char-Test ?  CR
Up-Char-Test @ 1 = [IF]
    : BETWEEN   1+ WITHIN ;                    ( x min max -- flag )
    : UP-CHAR ( c -- C ) DUP [CHAR] a [CHAR] z BETWEEN  BL AND  XOR ;
              SEE WITHIN  SEE BETWEEN  SEE UP-CHAR
[ELSE]
Up-Char-Test @ 2 = [IF]
    ( Eliminate nesting  UP-CHAR  BETWEEN  WITHIN  )
    : UP-CHAR                              ( c -- C )
      S" DUP [CHAR] a -  26 U<  BL AND  XOR " EVALUATE
    ; IMMEDIATE
```

Continued on page 27

Object-oriented programming in ANS Forth — code continues from preceding issue.

ARRAY.FTH — basic array classes

```
\ Classes for indexed objects    Version 1.0, 4 Feb 1997
\ Andrew McKewan                 mckewan@austin.finnigan.com

\ ============================================================
\ This is the base class for all indexed objects. It provides the
\ primitives that are common to all indexed objects.

:Class IndexedObj  <Super Object  CELL <Indexed

      \ ( -- addr )  Leave addr of 0th indexed element
      :M  IxAddr:   idxBase   ;M

      \ ( -- limit ) Leave max #elements for array
      :M  Limit:  limit   ;M

      \ ( -- len )  leave width of indexed elements
      :M  Width:  width  ;M

      \ ( index -- addr )  return then address of an indexed element
      :M  ^Elem:  ?idx  ^elem    ;M

      \ ( -- )  Indexed Clear: erases indexed area
      :M  Clear:  idxBase  width limit * ERASE  ;M

;Class

\ ============================================================
\ Basic cell array

:Class Array  <Super IndexedObj  CELL <Indexed

      :M  At:    ?idx  At4  ;M    ( index -- val )
      :M  To:    ?idx  To4  ;M    ( val Index -- )
      :M  +To:   ?idx  ++4  ;M    ( incVal index -- )

      \ Fill the array with a value
      :M Fill: ( val -- )  limit 0 DO  DUP I To4  LOOP DROP  ;M

;Class

\ ============================================================
\ X-Array can execute its elements.

:Class X-Array  <Super Array

      \ ( ind -- )  execute the cfa at Ind
      :M  Exec:  ?idx  At4  DUP 0= ABORT" Null xt"  EXECUTE  ;M
      :M  ClassInit:  ['] NOOP Fill: self  ;M

;Class

\ ============================================================
\ Basic byte array.

:Class ByteArray  <Super IndexedObj  1 <Indexed

      :M  At:    ?idx  At1  ;M    ( index -- val )
      :M  To:    ?idx  To1  ;M    ( val Index -- )
      :M  +To:   ?idx  ++1  ;M    ( incVal index -- )

      \ Fill the array with a value
      :M Fill: ( val -- )  idxBase limit ROT FILL  ;M

;Class
```

TESTER.FTH — Hayes' automated testing program

```
\ From: John Hayes S1I
\ Subject: tester.fr
\ Date: Mon, 27 Nov 95 13:10:09 PST

\ (C) 1995 JOHNS HOPKINS UNIVERSITY / APPLIED PHYSICS LABORATORY
\ MAY BE DISTRIBUTED FREELY AS LONG AS THIS COPYRIGHT NOTICE REMAINS.
\ VERSION 1.1
HEX

\ SET THE FOLLOWING FLAG TO TRUE FOR MORE VERBOSE OUTPUT; THIS MAY
\ ALLOW YOU TO TELL WHICH TEST CAUSED YOUR SYSTEM TO HANG.
VARIABLE VERBOSE
   FALSE VERBOSE !

: EMPTY-STACK   \ ( ... -- ) EMPTY STACK: HANDLES UNDERFLOWED STACK TOO.
   DEPTH ?DUP IF DUP 0< IF NEGATE 0 DO 0 LOOP ELSE 0 DO DROP LOOP THEN THEN ;

: ERROR         \ ( C-ADDR U -- ) DISPLAY AN ERROR MESSAGE FOLLOWED BY
                \ THE LINE THAT HAD THE ERROR.
   TYPE SOURCE TYPE CR                  \ DISPLAY LINE CORRESPONDING TO ERROR
   EMPTY-STACK                          \ THROW AWAY EVERY THING ELSE
;

VARIABLE ACTUAL-DEPTH                   \ STACK RECORD
CREATE ACTUAL-RESULTS 20 CELLS ALLOT

: {             \ ( -- ) SYNTACTIC SUGAR.
   ;

: ->            \ ( ... -- ) RECORD DEPTH AND CONTENT OF STACK.
   DEPTH DUP ACTUAL-DEPTH !             \ RECORD DEPTH
   ?DUP IF                              \ IF THERE IS SOMETHING ON STACK
      0 DO ACTUAL-RESULTS I CELLS + ! LOOP \ SAVE THEM
   THEN ;

: }             \ ( ... -- ) COMPARE STACK (EXPECTED) CONTENTS WITH SAVED
                \ (ACTUAL) CONTENTS.
   DEPTH ACTUAL-DEPTH @ = IF            \ IF DEPTHS MATCH
      DEPTH ?DUP IF                     \ IF THERE IS SOMETHING ON THE STACK
         0 DO                           \ FOR EACH STACK ITEM
            ACTUAL-RESULTS I CELLS + @  \ COMPARE ACTUAL WITH EXPECTED
            <> IF S" INCORRECT RESULT: " ERROR LEAVE THEN
         LOOP
      THEN
   ELSE                                 \ DEPTH MISMATCH
      S" WRONG NUMBER OF RESULTS: " ERROR
   THEN ;

: TESTING       \ ( -- ) TALKING COMMENT.
   SOURCE VERBOSE @
   IF DUP >R TYPE CR R> >IN !
   ELSE >IN ! DROP
   THEN ;
```

TEST.FTH — class test suite

```
\ test.fth -- Class testing    Version 1.0, 4 Feb 1997
\ Andrew McKewan               mckewan@austin.finnigan.com

HAVE _TEST_ [IF] _TEST_ [THEN] MARKER _TEST_

S" TESTER.FTH" INCLUDED
TRUE VERBOSE !
DECIMAL
{ -> }

\ ===================================================================
TESTING OBJECT CREATION

{ Var x -> }
{ 99 Put: x -> }
{ Get: x -> 99 }
{ Var y -> }
{ Get: x 1+ Put: y -> }
{ Get: y -> 100 }

: T1  Get: x ;
{ T1 -> 99 }

:Class Point  <Super Object

       Var x
       Var y

       :M Get:  Get: x  Get: y  ;M
       :M Put:  Put: y  Put: x  ;M

       :M Print:  Get: self  SWAP . .  ;M

       :M ClassInit:   1 Put: x  2 Put: y ;M

;Class

{ Point p -> }
{ Get: p -> 1 2 }
{ 3 4 Put: p -> }
{ Get: p -> 3 4 }

:Class Pixel  <Super Point

       Var color

       :M Put:  ( x y color -- )  Put: color  Put: super  ;M
       :M Get:  Get: super  Get: color  ;M

       :M Print:   Print: super  Print: color  ;M

;Class

{ Pixel pix -> }
{ 1 2 3 Put: pix -> }
{ Get: pix -> 1 2 3 }

TESTING ExecVec

ExecVec ex
{ Exec: ex -> }
: NINE 9 ;
' NINE Put: ex
{ Exec: ex -> 9 }

\ ===================================================================
TESTING LATE BINDING

:Class C1  <Super Object  <General

       :M Draw:   1 ;M

       :M Print:  3 0 DO  Draw: [ self ]  LOOP ;M

;Class
```

```
{ C1 o1 -> }
{ Print: o1 -> 1 1 1 }

:Class C2  <Super C1

       :M Draw:   2  ;M

;Class

{ C2 o2 -> }
{ Print: o2 -> 2 2 2 }

\ =================================================================
TESTING OBJECT POINTER

:Class C3  <Super Object

       C1 p1
       C2 p2
       Var pp1        \ pointer to p1
       Var pp2

       :M ClassInit:  Addr: p1 Put: pp1  Addr: p2 Put: pp2 ;M
       :M Print:      Print: [ Get: pp1 ]  Print: [ Get: pp2 ]  ;M
       :M Switch:     Get: pp1 Get: pp2   Put: pp1 Put: pp2 ;M
;Class

{ C3 o3 -> }
{ Print: o3 -> 1 1 1 2 2 2 }
{ Switch: o3 -> }
{ Print: o3 -> 2 2 2 1 1 1 }

\ =================================================================

TESTING ARRAY

10 Array a1
{ 6 2 To: a1 -> }
{ 2 At: a1 -> 6 }
{ 99 Fill: a1 -> }
{ 5 At: a1 -> 99 }

:Class C4  <Super Object

   10 Array a1

   :M At:  ( index -- value )  At: a1  ;M
   :M To:  ( value index -- )  To: a1  ;M
   :M Fill:  Fill: a1  ;M

;Class

{ C4 o4 -> }
{ 6 2 To: o4 -> }
{ 2 At: o4 -> 6 }
{ 99 Fill: o4 -> }
{ 5 At: o4 -> 99 }

TESTING X-ARRAY

{ 10 X-Array xa -> }
: ONE 111 ;
: TWO 222 ;
: THREE 333 ;
{ ' ONE 1 To: xa -> }
{ ' TWO 2 To: xa -> }
{ : T3 ['] THREE 3 To: xa ; T3 -> }
{ 1 Exec: xa -> 111 }
{ 2 Exec: xa -> 222 }
{ 3 Exec: xa -> 333 }
{ 4 Exec: xa -> }

\ ========================================

HAVE ALLOCATE [IF]
TESTING HEAP OBJECTS
```

The Computer Journal

Support for older systems
Hands-on hardware and software
Computing on the Small Scale
Since 1983

Subscriptions
1 year $24 - 2 years $44
All Back Issues available.

TCJ

The Computer Journal

P.O. Box 3900
Citrus Heights, CA 95611-3900
800-424-8825 / 916-722-4970
Fax: 916-722-7480
BBS: 916-722-5799

```
0 VALUE POBJ

{ Heap> Point TO POBJ -> }
{ Get: [ POBJ ] -> 1 2 }
{ 3 4 Put: [ POBJ ] -> }
{ Get: [ POBJ ] -> 3 4 }
{ POBJ Get: Point -> 3 4 }  ( class binding )
{ POBJ RELEASE -> }

{ 10 Heap> Array TO POBJ -> }
{ 6 2 To: [ POBJ ] -> }
{ 2 At: [ POBJ ] -> 6 }
{ 99 Fill: [ POBJ ] -> }
{ 5 At: [ POBJ ] -> 99 }
{ POBJ RELEASE -> }

{ : T10  10 Heap> Array TO POBJ ; T10 -> }
{ : T11  6 2 To: [ POBJ ] ; T11 -> }
{ : T12  2 At: [ POBJ ] ; T12 -> 6 }
{ : T13  99 Fill: [ POBJ ] ; T13 -> }
{ : T14  5 At: [ POBJ ] ; T14 -> 99 }
{ : T15  5 POBJ At: Array ; T15 -> 99 }  ( class binding )
{ : T16  POBJ RELEASE ; T16 -> }

[THEN]

CR .( Class tests complete )
```

ANS.TXT — ANS requirements

```
ans.txt
Version 0.2 alpha release 8/14/96     Andrew McKewan     mckewan@austin.finnigan.com
This program requires the following ANS Standard word sets:

CORE
     all

CORE EXT
     :NONAME ?DO ERASE CASE OF ENDOF ENDCASE
     TRUE FALSE HEX NIP PARSE PICK TO TUCK U> VALUE \

EXCEPTION (optional)
     CATCH THROW

FILE (ability to load text files)
     INCLUDED

MEMORY ALLOCATION (optional)
     ALLOCATE FREE RESIZE

TOOLS
     DUMP (optional)

TOOLS EXT
     [IF] [ELSE] [THEN]

The words from the EXCEPTION and MEMORY ALLOCATION word sets are optional and will
conditionally compiled using [IF] [ELSE] and [THEN].

No FILE words are used, but the source is distributed in text files so the system must have the
ability to load text files or convert the source to blocks.
```

MPE's Forth Coding Style Standard

[Portions of this document, including parts of some of the examples, were edited lightly by FD for publication in this format. —Ed.]

Headerless words

If there is a requirement to make a word or set of words internal or external, headered or headerless, then this requirement is to be identified before the definition of the word or words concerned:

```
\ - R.G. - 30/10/91 - word1

HEX

INTERNAL                \ or HEADERLESS
: WORD1
;
EXTERNAL                \ or HEADERS
```

If the headers are to be removed with a beheading mechanism, this directive should also be clearly identified:

```
HEADERLESS
: WORD1
;
HEADERS

BEHEAD
```

Vocabularies

If a vocabulary or context switch is to be made in the source code, the vocabulary should be the same at the end of a page as at the beginning. This means that if a new page is inserted at the end of a page, the search order and defining vocabulary will be known:

```
\ - R.G. - 30/10/91 - io words

ALSO IO DEFINITIONS
...
...
...
PREVIOUS DEFINITIONS
```

Definitions

The definitions will then follow on the page. The detailed layout standard for definitions follows later. There will be a blank line between the end of the last definition on the page and the end-of-page marker.

The length of a page

There is no fixed length to a page. However, as a guide, a page might fit neatly onto a sheet of paper when printed. A definition should be as short as possible. If many words are grouped on one page, then the page will be as long as is needed to accept the whole group of words.

Layout of a definition

It is acknowledged that a Forth definition should be as short as possible. This may be two or three lines, or it may be 15 or 20 lines. The actual size will depend on circumstances, but should always be as short as possible. Short words encourage the reuse of small code fragments, which leads to smaller code and to reliable code. It has been said that there are three types of procedure call:

- call by value
- call by reference
- call by text editor

Call by value means using the actual value required as the parameter to the word being called. Call by reference means using a pointer to the data needed for the word being called. Call by text editor means not making a call at all, but copying the code in the text editor itself. This last is to be avoided because

- the code gets bigger,
- the code is harder to maintain.

Header block comments

One method for writing a lengthy descriptive comment for a Forth word is to use a header block. This is a block of comments just above the start of the word, which describes the function of the word in detail. This is normally detail or description which would not fit well in the in-line comments down the right-side of the page:

```
\ this word ...
\ ...
\ ...
: word1            \ - ; does ...
 ...                      \ ...
;
```

MPE supports a long comment in the form of:

```
(( Function: foo
 Author: SFP
 Date: 19 May 95
 Inputs:
 Outputs:
  Algorithms:
 Changes:
 Description:
))
```

Microprocessor Engineering, Ltd.
Southampton, England

The contents need to be agreed and used. There is nothing worse than than a block comment with the template inserted but not filled out. Many users make these comments easier to find by using lines of asterisks.

```
(( ******************************
...
****************** ********** ))
```

The form and contents of header comments need to be defined centrally within your organisation. See also the section on change history.

Name and stack comment

The first line of a definition will consist of the start of the definition - either a colon (`:`) or a `CODE`, label (`L:` or `LBL:`), etc. and the name of the procedure. This will then be followed by the stack effect for the word:

```
: WORD1      \ n1 — n2 n3 ; description
 ...                        \ ...
;

: WORD2      ( n1 — n2 ; description )
 ...                        \ comment
;

CODE WORD2  \ n1 — n2 n3 ; description
  ...
END-CODE

L: PROC1     \ — ; description
LBL: PROC2   \ — ; description
```

The :, etc. will start at the very left-hand end of the line. There will be one space between this and the name of the word.

The stack comment and description will start some way across the line - but further towards the left and the word name than the in-line comments. *There will always be a stack comment.* Ensure the stack comment is correct. Within the stack effect, execution will be identified by one of the recognised marks:

—<R>   --<R> ->>

The "—" description is recommended, as formal source-scanning tools will look for this rather than the others. If it becomes necessary to also document the return stack effect, a "++" or "`R:`" should be used in place of the "—".

It is good practice to follow the stack effect with a short description of the action of the word - about three or four words:

```
: D*  \ d1 d2 — d3 ; double multiply
;
```

If there is a short description of the word, it should be separated from the stack effect by a semi-colon (`;`) or other obvious character. This will distinguish the description from a stack effect consisting of descriptive names for the stack items. Using a standard semi-colon, other formal tools such as source analyzers will correctly handle the source code and the comments.

It is also useful to establish conventions for naming items in stack comments. Clarity is the objective. For example the following have been seen to indicate an address and a length returning nothing:

```
a l
a l —
a/l —
addr len — ; the MPE house style
```

MPE house rule:

All words have stack comments and descriptions on the name line—no exceptions. Programmers who do not conform will be fired. Yes, this rule is important.

Indenting

The body of the word - the words it calls, or the assembler mnemonics it uses will be indented from the left hand end of the line. This indent will be uniform throughout the file, and will normally be two spaces. This brings the contents of a word in line below the name of the word. Control structures will be further indented. This is dealt with later on.

```
: WORD1      \ n1 n2 — n3 ; function to ...
  ...
  ...
;
```

Phrasing

Each line of code in a definition should constitute a readable and meaningful phrase. Forth should not be laid out so vertically that each line is individually meaningless. A single phrase will consist of enough code to perform some appropriate part of the application:

```
VAR @ 10 +<R>   OVER 4 <<<R>   SWAP 3 + BILL +!
```

Numbers

Many compilers allow the base to be specified as the number is typed:

```
#100        \ decimal 100
$100        \ hex 100 = decimal 256
%100        \ binary 100 = hex/decimal 4
```

If the compiler to be used supports this feature, then it is good practice to use it, as there can then be no mistake which number is meant at any time. If the compiler does not support the temporary base definition, then it is best to always prefix a hex number with a zero:

```
HEX
0100        \ hex 100 = decimal 256
0ADD        \ hex ADD = decimal 2781
ADD         \ the word 'ADD'
```

To be continued...

CRC-32

International Standard 32-Bit CRC

The subject of this article is calculation of the International Standard 32–bit CRC (cyclical redundancy check). It uses nonce–words, or throw–away definitions, to build a table to speed it up. The Standard Forth definitions of `::`, `;;`, and **ANEW** are discusssed in the accompanying articles, "Tool Belt #01" by Wil Baden and "The View from Goat Hill #01" by Neil Bawd.

[See Figure One.]

`update-crc-by-a-byte` and **CRC** are equivalent. However, **CRC** will be five to eight times faster. Since you want to do it to every byte you read or write, the speed is important.

Crc-Table and **CRC** are the only two permanent definitions. The other words given here are compiled, executed, and forgotten.

To use **CRC-32**, set the checksum to **TRUE** (all bits on) and, for every byte written or read, use **CRC** to update the checksum. Be sure you have set the file–access mode to binary on MS-DOS type systems.

When you've finished writing, write out the checksum, low byte to high byte.

When reading, include the last four bytes you read in the checksum you have been accumulating. If everything has gone right, the checksum will be 0.

Here is what's printed during compilation.

```
\ CRC-POLYNOMIAL is EDB88320

CREATE Crc-Table
HEX
00000000 , 77073096 , EE0E612C , 990951BA , 076DC419 ,
706AF48F , E963A535 , 9E6495A3 , 0EDB8832 , 79DCB8A4 ,
E0D5E91E , 97D2D988 , 09B64C2B , 7EB17CBD , E7B82D07 ,
90BF1D91 , 1DB71064 , 6AB020F2 , F3B97148 , 84BE41DE ,
1ADAD47D , 6DDDE4EB , F4D4B551 , 83D385C7 , 136C9856 ,
646BA8C0 , FD62F97A , 8A65C9EC , 14015C4F , 63066CD9 ,
FA0F3D63 , 8D080DF5 , 3B6E20C8 , 4C69105E , D56041E4 ,
```

Figure One.

```
: ANEW   >IN @ BL WORD FIND IF EXECUTE ELSE DROP THEN >IN ! MARKER
;
: ::     S" ANEW NONCE  : NONCE-DEF "  EVALUATE ; IMMEDIATE
: ;;     S" ; NONCE-DEF  NONCE "  EVALUATE ; IMMEDIATE
```

Figure Two.

```
( The International Standard 32-bit CRC. )

CREATE Crc-Table   256 CELLS ALLOT

MARKER CRC-TABLE-INITIALIZATION

( Define CRC-POLYNOMIAL from its coefficient terms. )
:: 32 26 23 22 16 12 11 10 8 7 5 4 2 1 0   ( ...)
      0 BEGIN                              ( ... poly)
            SWAP                           ( ... poly bit)
            DUP 32 = NOT
      WHILE
            31 SWAP -  1 SWAP LSHIFT  OR   ( ... poly)
      REPEAT                               ( ... poly bit)
      DROP                                 ( poly)
;; CONSTANT CRC-POLYNOMIAL                 ( )

CR .( \ CRC-POLYNOMIAL is ) CRC-POLYNOMIAL HEX U. DECIMAL CR
```

Wil Baden • Costa Mesa, California
wilbaden@netcom.com

Wil Baden is a professional programmer with an interest in Forth. To get the source for this article, send e-mail requesting "Stretching Forth: CRC-32 International Standard 32-Bit CRC."

Call for Papers: FORML Conference

New FORML dates...
...the week* before *Thanksgiving!

The original technical conference for professional Forth programmers and users.

19th annual FORML Forth Modification Conference
November 21 – 23, 1997

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California USA

THEME:

"Forth at the Millennium"

What are the challenges for Forth as we reach the Millennium? Will the year 2000 present problems for existing programs? Many organizations are asking for certification that software will work perfectly as we move to 2000 and beyond.

How will certification be accomplished? Encryption is required for more applications to keep transactions private. Proposals for incorporating encryption techniques are needed for current and future applications. Your ideas, expectations, and solutions for the coming Millennium are sought for this conference.

FORML is the perfect forum to present and discuss your Forth proposals and experiences with Forth professionals. As always, papers on any Forth-related topic are welcome.

Abstracts are due October 1, 1996 • Completed papers are due November 1, 1997

Mail abstract(s) of approximately 100 words to:
FORML, Forth Interest Group • 100 Dolores Street, Suite 183 • Carmel, California 93923
or send them via e-mail to FORML@forth.org

The Asilomar Conference Center combines excellent meeting and comfortable living accommodations with secluded forests on a Pacific Ocean beach. Registration includes use of conference facilities, deluxe rooms, meals, and nightly wine and cheese parties.

Guy Kelly, Conference Chairman • Robert Reiling, Conference Director

Registration and membership information available by...

phone: 408-373-6784
fax: 408-373-2845
e-mail: FORML@forth.org

mail: FORML, Forth Interest Group
100 Dolores Street, Suite 183
Carmel, California 93923

FORML Conference sponsored by the Forth Modification Laboratory, a Forth Interest Group activity.