$10 Volume XVI, Number 4 November 1994 December

FORTH DIMENSIONS

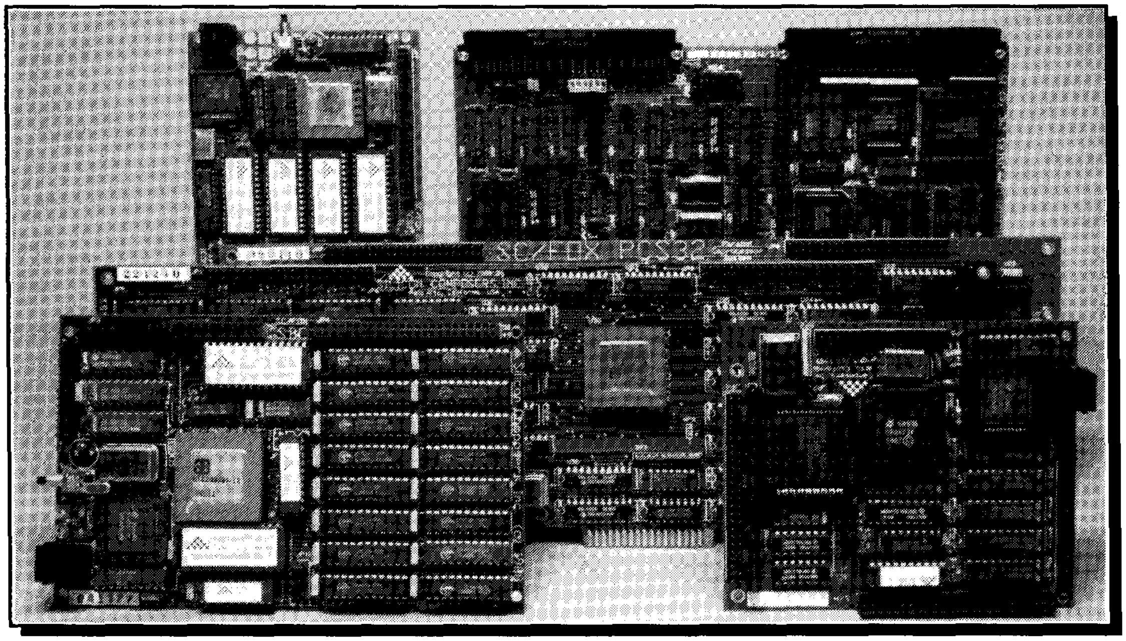
WLOAD for WRI Files

Jump & Execute Tables

HDTV Format Convertor

Object Code vs. Metacode

Algebraic Specification of Stack Effects



Contents

Features

Departments

Editorial

Maximum advantage: A comment on comp.lang.forth challenged whether the Forth community is really a community at all. Without consulting the dictionary, I'd define community as people who share mutual interests and mutual responsibilities. Our common interest here is obvious, but responsibilities? I've been considering the social contract we each are born into (and this jars my "inner anarchist"), the reciprocity that permits us to enjoy the benefits of a society. Yes, we must tolerate or even encourage diversity, but Forth proponents are still too few to survive much fragmentation. To prosper, we *must* call upon our ability to function as a community.

The impending (at press time) publication of the official ANS Forth document presents us with a new and potent opportunity. If—and only if—we pull together collectively and also do individually whatever we can, maximum leverage can be achieved.

* * *

A call for authors: We have received many kind comments this year regarding *Forth Dimensions.* One of my editorial tasks is to keep an interesting mix of ideas, techniques, application examples, and Forth stories to satisfy a diverse, international readership.

I must ask for your assistance. Help us to keep improving the quality of *Forth Dimensions* content (and help it to better represent your own viewpoints and expertise) by writing an article or letter to the editor. Remember that not all our readers are experts; your personal story of trial-and-error might be the perfect lesson and encouragement for someone else. And some Forth experts working on very exciting projects feel, by virtue of their hectic schedules, too preoccupied to write. So if you, perhaps too humbly, feel that your own Forth musings might not shed much light, consider interviewing one of the notables and writing about their work.

* * *

Bottom line: The Forth Interest Group has, for more than a decade-and-a-half, served as the primary focal point of Forth advocacy. Whether other sources of information about the language have been non-existent, strong, or erratic over this time, FIG has steadily provided access to expertise, vendor contacts, software archives, and printed literature. One can find things subject to constructive criticism, but enough cannot be said of the value that FIG's constant presence has brought to the whole Forth community.

Thanks to the support of its members, FIG has withstood the ebb and flow of programming fashions and the onslaught of worldwide recessions. Membership-related revenue is its financial lifeblood. The quantity and quality of services provided to members, and the number of projects that can be undertaken to promote Forth, are quite directly related to the number of members in this non-profit organization. (If you know a way to develop substantial income from other sources, please contact FIG and get involved!)

Base-line participation in the Forth Interest Group is simply by having an active membership. And the most direct way to ensure FIG's continued ability to serve is by encouraging co-workers, customers, students, fellow chapter attendees, and even your employer to join FIG and to receive *Forth Dimensions.* If it is appropriate, buy someone a gift membership. Don't underestimate this simple form of participation—our organization needs financial support, just as we need written contributions to these pages. And FIG needs new members every year, just as Forth needs new users and markets.

Consider the mutual interests and responsibilities of our community; and think of the influence a strong community can have in Forth's penetration into existing markets and creating new ones.

—Marlin Ouverson
ouversonm@aol.com

Forth Dimensions
Volume XVI, Number 4
November 1994 December

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

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

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

The Forth Interest Group

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

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

Letters

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

Fractional Math

Dear Marlin,

I've been noting with interest the increasing number of articles that deal with fractional mathematics, such as two articles in *Forth Dimensions* XVI/2. This has gotten me thinking about the possibility of synthesizing some of the individual pieces into a comprehensive fractional-math package, preferably one that would be portable across a large number of platforms. I've also thought of possibly using cordic functions to help generate transcendental functions for small values, then using various identity formulas to scale them up to larger values. I'm thinking of presenting the results, including code, for an article in *FD* within the next year.

I would like to make some additional comments, geared specifically toward the article, "Convert Real Numbers to Fractions," by Walter J. Rottenkolber. The continued-fraction technique that he discusses is presented in most college-level courses on number theory. One point that must be brought up when generating continued-fraction approximations is that the results can never be more accurate than the original value used to approximate the desired value. The predicted accuracy of each fractional approximation A/B to some value X, where A/B is generated by the continued-fraction algorithm, is always |X-(A/B)| < 1/(B^2). If the approximation's theoretical accuracy exceeds the accuracy of the figure you give it vs. the desired value (as in using 3.1416 for pi), you should always consider the returned fraction(s) to be suspect from that point on.

A second point is that the continued-fraction algorithm also gives a fast way for reducing any fraction to its lowest terms. If a fraction P/Q is given to the algorithm, and it is run until the residual value is zero, then the final fraction A/B generated is P/Q reduced to lowest terms. For example, if you start with the fraction 68/255, you will get back 4/15.

Finally, it might be interesting to look at the possibilities raised by "quadratic" integers and fractions, which are also discussed in number theory. In particular, numbers of the form (A+(B*sqrt(N)))/C may be useful in solving certain types of geometrical problems, especially since the sines and cosines of certain angles, such as 30, 45, and 60 degrees, can be expressed as "quadratic" fractions.

Along similar lines, I'd be interested in seeing if there is any interest out there for a comprehensive package that would handle arbitrarily large integers and fractions; such a package might be useful for mathematics research, for example. The number values themselves would be represented as strings of cells; a heap manager would be necessary. While such a package could be put together such that it would have a high degree of both system and dialect independence, it would be too involved for an article; rather, it would become a small book! (It would possibly have a disk for MS-DOS included.) Interested readers can contact me via e-mail at the address listed at the end of this letter.

My last major remark is along completely different lines, and is directed at the article, "Forth Nano-Compilers," by Veil and Walker. It appears that they are approaching problems similar to the ones that I discussed in my own article, "Optimizing '386 Assembly Code" (XV/6), but from a somewhat different perspective. It is interesting to compare notes on the parallel techniques being used to solve similar types of problems.

And finally, last but not least, one minor note: could you start regularly publishing a contact point, such as an e-mail or "snail-mail" address, where each author could be reached? Such contact points would give interested readers the ability to follow up *directly* on each article.

Yours,
David M. Sanders
PsiqFour@aol.com

Thank you for your remarks, David. As to publishing authors' addresses: recently, primarily for reasons of privacy, we generally have not published addresses unless they are a business location or a post office box. We encourage authors to include e-mail addresses, although it is their decision. We like it when an article or letter inspires dialog—just don't leave FD *and its other readers out. Please send copies of interesting amplifications, corrections, and even disagreements (see the following letters) so that we all can enjoy the discussion. —Ed.*

The Essence of Forth?

In "The Essence of Forth," Randy Leberknight and Dennis Ruffer describe a cumbersome procedure for locating a particular Forth definition within a hierarchy of files on a Unix system, contrasting this procedure with the simplicity of the "LOCATE <wordname>" command in the Forth, Inc. editor.

In fact, the two most common Unix editors (vi and GNU EMACS) each have built-in commands, even more streamlined than LOCATE, for automating the process.

For example, using the vi editor, you can either:

a) Type "vi -t <wordname>" to start the editor with the cursor positioned on the definition of that word within the correct file, or

b) Position the cursor anywhere within an occurrence of the word you wish to locate and type "^}" (control-right brace), to switch the current edition session to the correct file and position, or

c) Type ":ta <wordname>" to do the same thing as (b), if the cursor doesn't happen to be anywhere near an occurrence of that word.

Having moved around through several such "tags," you can backtrack to any level ("pop the tag stack") by typing "^]" as many times as necessary.

GNU EMACS has even more powerful capabilities—for example, it can do directed search-and-replace, spanning exactly the files comprising a project. In fact, GNU EMACS has all the features that the article attributes to Forth, Inc.'s editor, and many more besides.

At a more fundamental level, I question the overall premise of the article in question. The article, entitled "The Essence of Forth" and beginning with the rhetorical question about the key to Forth's productivity, is mainly a description of Forth, Inc.'s editor. If Forth, Inc.'s editor is indeed the essence of Forth and the key to Forth's productivity, what have the rest of us Forth enthusiasts—who don't have access to said editor—been doing all this time?

Certainly Forth, Inc. has a powerful and useful editor, but is that editor the "essence of Forth"? I don't think so.

Mitch Bradley
President, FirmWorks
480 San Antonio Rd., Suite 230
Mountain View, California 94040
wmb@firmworks.com

Spaghetti in Any Language

Walter J. Rottenkolber's satire "Switch in Forth" was very amusing. But it is not necessary to mimic C for spaghetti code: Standard Forth will do as well on its own. *[See Figure One.]*

Wil Baden
wilbaden@netcom.com

Figure One. Baden's `switch-example`.

```
: but 1 CS-ROLL ; IMMEDIATE

: switch-example
 DUP 1 = IF DROP   one ( C: orig)
 BEGIN but   three ( C: dest orig)
 ELSE DUP 2 = IF DROP but THEN two
 ELSE DUP 3 = IF DROP  three ( C: dest orig orig)
 ELSE DUP 4 = IF DROP but THEN four
 ELSE [ 2 CS-ROLL ] DROP AGAIN  ( C: orig orig)
 THEN THEN    ( C: )
;
```

In ThisForth, `switch-example` can be written:

```
: switch-example
 CASE 1 OF   one ( C: orig)
 BEGIN but   three ( C: dest orig)
 ELSE 2 OF but THEN  two
 ELSE 3 OF   three ( C: dest orig orig)
 ELSE 4 OF but THEN  four
 ELSE [ 2 CS-ROLL ] DROP AGAIN  ( C: orig orig)
 ESAC     ( C: )
;
```

Switch Suggestions

Dear Marlin,

I was intrigued by Walter J. Rottenkolber's implementation of a C-like "switch" in *FD* XVI/3. I've thought about this particular construction from time to time, but was reluctant to implement it because I couldn't think of a situation in which it would be useful. Nevertheless, Rottenkolber's attempt moves me to offer some suggestions—I think improvements—in syntax, orthography, and implementation.

Implementing "break" as `EXIT` could hardly be simpler, though it makes C-like nested "switches" impossible, but with `BREAK` as an alias of `EXIT`, it seems appropriate to make `SWITCH:` a defining word, so I include `[COMPILE] :` in `SWITCH:` and define a `;SWITCH` which includes `[COMPILE] ;`. This makes each "switch" a named Forth word, and avoids the error Rottenkolber warns about in his ante-penultimate paragraph. And a nested "switch" can be defined as a word and included as a factor in the "switch" it nests in.

The word `<DEFAULT` in Rottenkolber's implementation seems awkward to me. I was able to dispense with it, and also with `DEFLG`. In their place, I use two *compile-time* variables which allow me to avoid the run-time penalties of variable checking and of the `IF` in `SWITCH:` that always jumps to the first "case" (`CASE'` or `DEFAULT`, whichever is first).

Rottenkolber uses three words for each case: `CASE'`, some variation of `=;;`, and `;;` to end it. I use two, which I spell `CASE(` and `)IS` (other versions of the second word, corresponding to `<;;`, `>;;`, `RANGE;;`, etc., are trivial to add), and am able to dispense with the `;;`. Rottenkolber's version makes a common C idiom awkward. The fragment

```
...
case 1:
case 2:
case 3:
case 4: dog(); break;
...
```

would have to be coded as

```
...
CASE' 1 =;; ;;
CASE' 2 =;; ;;
CASE' 3 =;; ;;
CASE' 4 =;; DOG BREAK;
...
```

I've been able to implement it as

```
...
CASE( 1 )IS
CASE( 2 )IS
CASE( 3 )IS
CASE( 4 )IS DOG BREAK
...
```

which I think is cleaner and closer to the

(Continues on page 32.)

WLOAD For WRI Files

Hank Wilkinson
Greensboro, North Carolina

Microsoft Windows comes with two word processors: the simple Notepad generating straight ASCII files, and the more complex Write creating pleasing documents. Either serve as a Forth source code editor. Notepad's ASCII files pose no difficulty, and the "Save As..." option in Write can saves files as straight ASCII, too. Write's attributes easily allow good documentation of Forth code, and a little understanding of the Write file's structure allows Forth to load WRI files without using the "Save As..." option.

Write tries to give the ".wri" extensions to file names, and these WRI files hold pictures and text. WRI documents allow easy formatting and editing, and print easily too. The command WLOAD, described in this article, eases creation of clear documentation by loading WRI files. Glen Haydon's article, "Formatting Source Code" (*Forth Dimensions* X/6) provided guidance for the code presented here.

Glen's programming switches { and } allow easy documentation in a novel way. Reversing the relation between code and comment, Forth expects comments until explicitly told code follows. Notice this is backwards from the normally terse state, encapsulating comments with parentheses inside a line, or backslashing them to the end of the line.

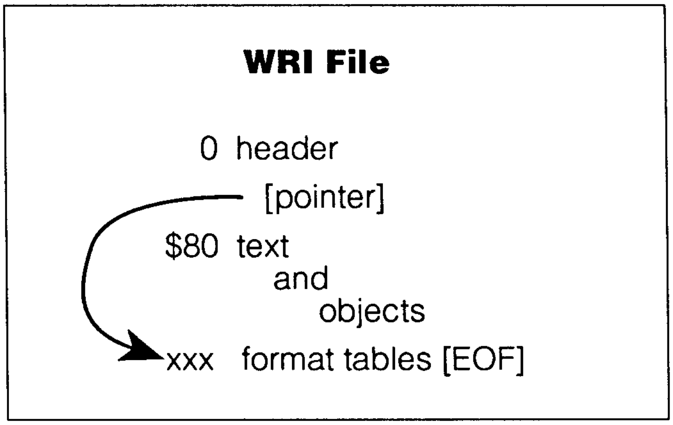
Some things are simple until you try them out. Conversely, some things are easier than you think.

Simple motivation prompts this article. The one time I received a letter about a Forth program I wrote, the program used Glen's switches. Finding my program on a bulletin board, the reader praised me—in writing—for such well-documented Forth. Contrary to widely held general "knowledge," Forth may be documented so well people compliment you!

Even if you do not like this method of coding, documentation forges a fundamental link in programming. Communicating to oneself and others often proves more difficult than communicating to the machine. We struggle to understand the source code of a working program which, by virtue of its execution, communicates fine to the machine! Documentation deserves all the facilities our word processors offer, including pictures when helpful. There are times you may need to use WLOAD, or something like it.

The complexity of a Write file made this job difficult to figure out, but *Inside Window File Formats* by Tom Swan (Sams Publishing, 1993) helped. Tom's book, nicely written for C programmers, explains various formats. One chapter exclusively discusses the Write file, which we condense.

A WRI file consists of a 128-byte header (containing a pointer to the document's format tables), followed by text and objects forming the contents of a document, and finally the document's format tables.



Text (including page headers and footers) and data objects begin on byte $80 (i.e., the 129th byte) of the file, right after the header. We make no tests on the header because Write generates our files, but simply extract the pointer to the format tables. Below, we rename our pointer "text & object size in bytes" and show its location.

Windows 3.1 text consists of the extended Windows character set, plus control characters like the carriage return, page break, and so on. Objects, signified by their first byte, allow different kinds of things besides text. Currently, objects exist as either the $E3 or $E4 type.

We ignore the $E3 type of object here because I don't use them. Object linking and embedding, or OLE, creates

WRI File Header

```
0    $31 or $32
1    $BE
.
.
.
F  }
10 } text & object
11 } size in bytes
12 }
.
.
.
7F   end of header
```

OLE Header

```
0    $E4
1    $BE
.
.
.
10 }
11 } object size
12 } in bytes
13 }
.
.
.
1E } object header size
1F }
.
.
.
```

the $E4 type. Copying and pasting from OLE-aware applications into WRI documents generates $E4 objects. Many applications are OLE-aware: Windows Paintbrush (drawings), for example, and of course Write.

Hitting $E4 in a WRI file, our code reads pointers from the object's header, telling our code the total object size.

Shown above, byte 16 (the 17th) of the object's header starts a four-byte number containing the object's length in bytes. Bytes 30 and 31 (the 31st and 32nd) tell how long the header itself is. Adding the object size to the header size determines the total size of the object to ignore.

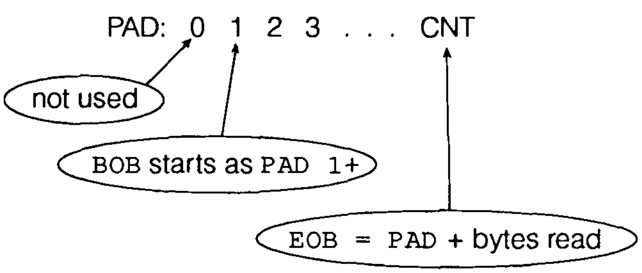
Write documents may get very large, especially with pictures. I periodically use some exceeding 250K, and the test file used for this code is over 280K. HS/FORTH's 64K source code size limit suggested the "strip Forth code to a dummy file" approach. This technique still has the 64K segment limitation, but only on the final stripped Forth source code size, not the initial WRI file's size. With Forth code greater than 64K, simply use more than one file.

WLOAD works like this:

```
Open WRI file and dummy file
Aim pointers
    Until end of text and object area
        Look for { or $E4
            If { found, copy text to dummy until } found
            If $E4 found, ignore entire object
Close files
Pass dummy file to Forth and LOAD
```

Our buffer design reflects pathological fear of reading past the end of a file, the end of a buffer, or simply losing track of a single byte. Using the same buffer for reads and writes avoids moving text around. Below shows the buffer and pointer scheme.

Disk I/O Buffer



BOB points to the current byte. Obviously, BOB 1+ reflects the next byte. EOB BOB - 1+ computes the number of bytes left in the buffer. EOB BOB < yielding true means an empty buffer. To compute the number of bytes processed, first we save the current value of BOB, do our work inside the buffer, then subtract the new current value of BOB from the old one.

The disk writing routines use the HS/Forth word WRITEH which needs a paragraph (provided by LISTS @) and offset address, a count of bytes to write, and the file's handle. Reading uses the unique HS/Forth command N@H, needing only a count and handle because N@H places the bytes at PAD 1+. I tested this definition of N@H, if you lack one.

```
: N@H ( count handle -- PAD_1+ )
  >R >R LISTS @ PAD 1+ R> R> READH DROP
  PAD 1+ ;
```

```
 0 VAR FALSE
-1 VAR TRUE
CREATE RD$ 128 ALLOT \ read file name holder
0 VAR RD-H           \ read handle
CREATE WR$ 128 ALLOT \ write file name
0 VAR WR-H           \ write handle

0 VAR CNT            \ # of bytes read into buffer
0 VAR BOB            \ offset to current byte in buffer
0 VAR EOB            \ offset to end of buffer
0 S->D DVAR FSZ      \ main file reading counter
128 VAR BSZ          \ buffer size

\ fills buffer with BSZ or less bytes. # read returned by CNT
\ FSZ shows remaining bytes, BOB & EOB set
: GETBUF ( -- )
\ test for EOF
FSZ D0= IF 0 IS CNT PAD 1+ IS BOB PAD CNT + IS EOB EXIT THEN
\ make sure there are BSZ bytes left, adjusting CNT
FSZ BSZ S->D
D> IF FSZ BSZ M- IS FSZ
      BSZ IS CNT
   ELSE FSZ DROP IS CNT
        0 S->D IS FSZ THEN
\ read the file, aim pointers
CNT RD-H N@H IS BOB PAD CNT + IS EOB ;

\ reads one byte onto stack, pointing BOB to next byte
\  or filling buffer if necessary
:  GETBYTE (  -- byte )
EOB BOB < IF GETBUF THEN \ fill buffer if necessary
CNT 0= IF FALSE EXIT THEN \ exit with FALSE upon EOF
BOB C@ BOB 1+ IS BOB ; \ get byte and adjust pointer

\ writes bytes to file
: PUTBUF   (  PAD_address count  --  )
  >R >R LISTS @ R> R> WR-H WRITEH DROP ;

\ ignore a given # of bytes in read file
: IGNORE ( n -- )
  EOB BOB - 1+ OVER U<
  IF
     EOB BOB - 1+ -
     BEGIN
      GETBUF CNT OVER U<
     WHILE CNT - REPEAT
  THEN
  BOB + IS BOB ;

\ given an object, find size and strip it
: STRIP-OBJ   (  --  )
16 IGNORE
GETBYTE GETBYTE 256* + GETBYTE GETBYTE 256* + \ double on
stack
10 IGNORE
GETBYTE GETBYTE 256* + 32 - M+
\  convert to single to IGNORE
BEGIN DUP WHILE -1 IGNORE -1 0 D- REPEAT
DROP DUP IF IGNORE ELSE DROP THEN ;
```

One advantage of Forth is its ability to build working prototypes to test. Some things are simple until you try them out. Conversely, some things are easier than you think. Forth allows building many prototypes, learning the best approach from experience. Though I have written code like this before, this is my first time for Windows and HS/Forth. The code shown with this article appears in a different form from earlier methods used.

I point you now to the code. First come various holders for data and values used throughout the routines. GETBUF, GETBYTE, and PUTBUF access the files, keeping all pointers aligned. IGNORE moves pointers past a given single-number count of bytes. Given the $E4 byte from the input stream, STRIP-OBJ computes an object's size from its header, handling double numbers for IGNORE.

IS}? performs the job of looking inside a buffer for }. (This punctuation and spelling are messing with my grammar checker!)

{ uses IS}?, copying a file until }. LOOK decides to call { or STRIP-OBJ. DO-WORK looks inside a file for the two characters { and $E4, calling LOOK when encountered.

EXTRACT-FORTH, then, places only text

```
\ looking for } inside buffer
: IS}? ( -- flag ) \ TRUE, } found, BOB points to it
BEGIN
  BOB C@
   ASCII } = IF TRUE EXIT THEN
  BOB 1+ IS BOB
EOB BOB < UNTIL
FALSE ;

\ copy until }
: {  (  --  )
CNT 0= IF EXIT THEN \ do nothing if empty
BOB EOB = IF GETBUF
          ELSE BOB 1+ IS BOB THEN
BEGIN
  BOB IS}? IF BOB OVER -    PUTBUF EXIT
          ELSE EOB OVER - 1+ PUTBUF THEN
  GETBUF
  CNT 0= IF EXIT THEN
AGAIN ;

: LOOK  (  --  )
BOB C@ ASCII { = IF { ELSE STRIP-OBJ THEN ;

: DO-WORK  (  --  )
GETBUF
BEGIN
  CNT 0= IF EXIT THEN \ EOF, so quit
  BOB C@ ASCII { =  \ looking for {
  BOB C@ 228 =      \ looking for $E4
  OR IF LOOK THEN   \ if found, do something
  \ next character
  BOB EOB > IF GETBUF ELSE BOB 1+ IS BOB THEN
AGAIN ;

: EXTRACT-FORTH (  --  )
$" d1" WR$ $! \ dummy name to put Forth
RD$ OPEN-R IS RD-H WR$ MKFILE IS WR-H \ open files
14 RD-H N@H DROP \ get to end_of_data pointer
\ set FSZ to end of data
1 RD-H N@H C@ 1 RD-H N@H C@ 256* +
 1 RD-H N@H C@ 1 RD-H N@H C@ 256* + IS FSZ
128 18 - RD-H N@H DROP \ ignore rest of header
FSZ 128 M- IS FSZ \ tell FSZ header was read
\ now extract Forth from text and objects
DO-WORK
\ and close files
WR-H CLOSEH
RD-H CLOSEH ;

\ Use: WLOAD filename.wri
\ ( FLOADs Forth code in filename.WRI )
: WLOAD
   BL TEXT
    PAD RD$ $!
    EXTRACT-FORTH
     $" D1" <MFLOAD> ;
```

found between curly braces inside a dummy file named D1. The action of EXTRACT-FORTH handles the files, sets the data size FSZ from the WRI header, and jumps past the rest of the header. After the DO-WORK call, EXTRACT-FORTH closes the files.

WLOAD reads a file's name from the input stream, stores it in a string variable, and begins the extraction. The syntax

```
WLOAD path\filename.ext
```

requires no quotes on the filename. The dummy file with the code passed to HS/Forth's <MFLOAD> receives a check for file size before loading.

Three problems exist with WLOAD. Mainly, the WRI file has to be closed before you WLOAD it, or you get a SHARE abort from MS-DOS. Having to close the file reduces immediacy. Also, the routines also lack any parsing, like the BL WORD phrase, so neither the left- nor right-brace may be used in your code or document, except as switches. Finally, you can't WLOAD from within a WLOAD because you would overwrite D1 (but you may FLOAD inside a WLOAD). With those exceptions, these routines WLOAD Windows WRI files from Forth.

MuP21—Evolution of a Forth Chip

C. H. Ting
San Mateo, California

In the beginning, Chuck Moore designed the NC4000 chip.

It was 1984, the chip worked, and it was marvelous to behold. It was a 16-bit chip running at 5 MHz, it executed one to five Forth instructions per clock cycle, and it averaged about 12 MIPS. At the time, IBM was still struggling along, limping from XT to AT.

The only trouble was that the NC4000 worked so well the first time out of the foundry (Mostek in Colorado), that Novix decided to market it as a real product even though it was known that there were a few bugs: for instance, the interrupt could disrupt a two-cycle memory access, and you could not multiply one number with an odd multiplier. Novix tried to fix the bugs without Chuck, and the fixes were worse than the bugs. Finally, Novix, under financial stress, sold the NC4000 patent to Harris.

Harris threw in resources only a big company could muster to fix the bugs and added a number of enhancements. It added the on-board data and return stack, one-step multiplier, counter-times, and an interrupt controller, and built the Real Time Express (RTX2000) chip. For two years, Harris spared no efforts in promoting it, and the RTX2000 began to penetrate into lots of new applications. Just then, Harris decided to disband its digital division, which hosted the RTX project. Even now, though, Harris makes RTX chips for whoever needs them; but the marketing thrust has gone.

So we had a handshake agreement to build the P20 chip.

In the meantime, a fellow named Russell Fish came to the San Francisco area with a mission to revive Apollo, a workstation manufacturer out east which was eventually bought by H-P. Russell was introduced to Chuck, and they conspired to build the successor to the NC4000. It was a 32-bit microprocessor, code named ShBoom. Chuck designed and laid it out at the OKI Design Center in the Silicon Valley, and the prototype was built in the OKI main plant in Japan. The prototype worked well, but the partnership between Chuck and Russell fell apart. Chuck was left with two ShBoom prototype chips. He experimented with a new concept, designing the next generation of ShBoom on ShBoom itself.

After designing two chips—one with gate arrays at Mostek, and one with custom ASIC at OKI—Chuck was convinced there ought to be a better way to design chips. The chip design software packages were cumbersome, and, by insisting on useless rules and protocols, they tended to prevent one from optimizing a design. It was totally unreasonable to spend hours and hours, even on the fastest and largest mainframe computers, to simulate a single instruction. Chuck was dreaming about a CAD system he could use to design a chip at home. He started implementing the concept on the ShBoom and demonstrated the CAD system at several Forth conferences. People were generally impressed, but nobody had enough faith in him to make it happen.

That brought us to 1990. At the time, Orbit Semiconductor introduced the Foresight multi-project wafer processing service. It could produce 12 prototype (TINY) chips at a cost of $1500, with die size of 2.4 x 2.4 mm packaged in a 40-pin DIP case. I thought I could afford to pay for this service if Chuck were to design a microprocessor on that die.

Chuck looked at the information Orbit provided and decided he could not fit a 32-bit microprocessor in that TINY package—with 40 pins, the best he could do was a 20-bit microprocessor. A 20-bit microprocessor could be a nice design because it would match very neatly with the 1Mx4 DRAM memory chips which started to appear on the market. He was anxious to develop a new microprocessor that would become a platform for the CAD system on which he wanted to build future chips, in place of the two ShBoom chips he had.

So we had a handshake agreement to build the P20 chip.

The chip was officially called MuP20, because it has multiple processors integrated in a single package. Besides the 20-bit microprocessor with two stacks, it also has a memory coprocessor which allows the chip to talk directly to DRAM and SRAM chips, and a video coprocessor which generates live NTSC color TV signals from image data stored in DRAM.

As usual, it takes lots more than what you anticipate to accomplish anything. P20 started out as a three-month project, and it drags on for three years. The biggest problem was that Chuck had to debug his design and his tools at the same time. Time and again, we were at a loss, wondering who was telling the truth: the silicon or the simulator.

The first major change was that Chuck moved the CAD system from ShBoom to a '386 PC. The ShBoom system Chuck built had only one megabyte of memory and it was too small for P20. Having the CAD system in a standard '386 PC was very reassuring because we didn't have to worry about anything happening to the ShBoom chips. The CAD system could be backed up conveniently. Chuck built the '386 OK system as the operating environment for the CAD system, which he now called OKAD.

The '386 OK system took advantage of DOS capabilities and the protected mode of the '386 microprocessor. It is a graphical user interface to the CAD design system. The user can lay out a chip, edit the design conveniently, and simulate the chip functions, through a set of menus controlled by seven keys on the regular PC keyboard. It seemed to be a great waste to use a 101-key keyboard for this purpose, but the conventional keyboard is more rugged and much easier on the fingers than the many versions of the seven-key keypad Chuck was experimenting with. (One such keypad almost incapacitated Chuck's right arm.)

The OKAD system uses a tiled structure to hold a chip layout. Each tile was four microns on a side. It can be programmed to represent a transistor, and electric connections within and between the diffusion, poly silicon, first metal, and second metal layers. The tiled structure has many advantages. It allows silicon logic and connections to be specified without ambiguity, and it can be scaled conveniently as the CMOS technology moves rapidly from microns to submicron geometry. Another advantage is that the cell structure enforces many of the design rules automatically. Hence, the layout will be correct by design, ideally.

Chuck wondered what Forth programmers will think of a Forth machine without SWAP and OVER.

The second major change was that all the registers and stack elements were enlarged from 20 bits to 21 bits. The extra bit served two purposes. In ALU operations, it becomes the carry bit to accommodate extended-precision math operations. For memory access, it distinguishes DRAM from SRAM and I/O space.

A third major change was in the timing circuitry. Originally, Chuck used a set of counter registers to provide proper timing signals to the memory coprocessor. Then he thought the analog delay circuits were simpler, more efficient, and more elegant. He changed the design, using some weak transistors to charge big capacitors to generate the desired timing signals for all the components in the chip.

We went to Orbit Semiconductor many times. It was not until the sixth try that we got functioning chips. Even then, it was not easy to coerce the chip to talk. In that prototype, the SRAM-accessing circuit ran too fast. It read one instruction from SRAM and executed it. However, before the SRAM could supply the next instruction, the CPU read again and executed the same instruction a second time. Hence, each instruction at an even address was executed twice, while those at odd addresses were ignored. After pondering this strange behavior for a week, Chuck was able to write a boot routine—in which every instruction was repeated twice—and got the chip to boot into DRAM. Once the chip ran in DRAM, everything seemed to work as expected.

In this prototype, only three registers on the data stack could be accessed. All routines had to restrict stack usage to three or less. It was a great handicap, but it did not prevent Chuck from writing some impressive demonstration routines showing off the video coprocessor. It generated brilliant color graphics on TV monitor screens. Without an ASCII character set, he programmed P21 to dump its memory on screen in binary, using long sticks for ones and short sticks for zeroes. If he arranged the short sticks properly, the memory dump looked remarkably similar to hexagrams from I Ching.

Another problem was that OVER did not work. In the earlier designs, Chuck had already eliminated SWAP, because to swap data between the top and the next register on the data stack, one would need another intermediate register. The extra register was deemed unnecessary, and SWAP was eliminated from the instruction set. Now, for some reason, OVER did not work either. We decided to eliminate OVER from the instruction set as well. Chuck wondered what Forth programmers will think of a Forth machine without SWAP and OVER. Nevertheless, Chuck was able to code the entire OK system in P21 without these two instructions, and using only three elements on the data stack. Maybe we don't need SWAP and OVER after all.

The seventh prototype was delivered in early 1994. It worked much better. The SRAM timing was fixed, so we do not have to used the very elaborate boot routine. The data-stack accessing problem was also fixed, and all five registers are now available.

MuP21 is a reality, although we are still anxiously waiting for it to be produced in volume.

Dr. C.H. Ting, a long-time, noteworthy figure in the Forth community, may be reached via e-mail at Chen_Ting@umacmail.apldbio.com or by fax at 415-571-5004.

F83

Jump and Execute Tables

for Directing Program Control Flow

Walter J. Rottenkolber
Mariposa, California

Selecting one of many options is common in programs. You could use multiple IF ELSE THEN branches but, after a few levels, the branch code begins to obscure the select logic. The CASE and switch statements simplify the syntax so that the function of the code is more apparent. So widespread are their use that many new Forthwrights may be unaware that alternate selector words are available; namely, jump and execution tables.

I use Laxen and Perry's F83, an indirect-threaded Forth that follows the Forth-83 Standard. Since Forths vary in implementation, not all the comments I make may apply to your system. So test the code before committing yourself to it.

We can use tables as selectors because, in Forth, the distinction between data and functions is not as sharp as in other languages. With ' (tick), we can get the code field address (CFA) of a word. This is also known as the compilation or execution address. It contains the address of the code machinery that will process the contents of the parameter field. If it is executed, the word is run.

Each takes no more than a screen of code to implement, and can be easily modified.

Try this experiment from the command line:

```
: t$  cr ." Hello Forth World" ;
' t$  execute
```

If all goes well, the string should print out.

In most of the following words, the CFA is in an array and we start with the location address. So the CFA first has to be fetched from the array before being executed. Since this is a common practice, some Forths have the word PERFORM that combines @ EXECUTE.

Jump Tables

Screens two, three, and four provide examples of jump tables. These are single-dimension arrays (or vector or matrix, take your pick) in which the data are the CFAs of the words to be run.

In the first, separate words hold the data and the run code. You can use CREATE outside a colon definition, but you must then arrange to compile the data into the parameter field. In this example, the words] and [turn the compiler on and off. It's important that only regular Forth words be compiled this way. Numeric data would be compiled along with (LIT) by the interpreter, and so would be in a form not accessed directly by the jump routine.

The run code calculates the address of the desired word from the index value and the array's base address. The index value is doubled, to allow for the fact that word addresses are two bytes long, and then is added to the base address. This gives the array address we need to fetch the word address. Since we can jump directly to the word without scanning through the preceding words, this method is called, naturally, a jump table.

An alternate method of compiling is to use ' (tick) to get the code field (compilation) address, and then compile it with , (comma), e.g., ' CHAR , and ' P-IN , and etc. Screen two uses this method in the DO LOOP after CREATE. The number of elements on the stack controls the compile loop, and is also saved for the run-time index limit check.

CASE: (in screen four) is one of those tricky "can you top this" Forth words. CONSTANT is equivalent to CREATE , . This generates a new header and also saves the number of elements on the stack for MAP to use. HIDE prevents recursion, and] turns the compiler on to compile the words. MAP does error checking to ensure the index is in range before calculating the pointer address. PERFORM fetches the word's code address and EXECUTEs it. ; (semicolon), among other functions, REVEALs the word and turns off the compiler.

The use of the CASE: name for a jump table is not far-fetched. Some Pascal compilers detect if the index values are contiguous and generate a form of jump table instead of the usual nested IF THEN branches.

The jump tables in screens three and four use the first cell in the array to hold the count of words listed to test for an out-of-range index. I changed the original code to make the comparison with an unsigned operator (U<). If a signed comparison is used, a negative number would pass the test, even though it is equivalent to an unsigned number above 32767. After the address calculation, the jump could be to anywhere. Since I planned to use these tables with EXEC.TABLE1, I also ensured that they would detect a -1 (65535) as an out-of-range value (some tables don't).

Jump tables are useful, but take care. The selector (index) values and word list must be contiguous. If the word list has a gap, a default word such as NOOP must be inserted to pad the list. Also, the Forth standard has indexes start with zero and go to n-1. If there are five words to select, the index ranges from zero to four.

Execution Tables

Execution tables can be implemented as two tables, or as two-dimensional arrays. First, a list is scanned for a match to the test value. Its position is then used to locate the desired word.

A two-table execution table is described in screen five. The first table is scanned for a match value, and its position in the table is placed on the stack. A no-match is marked by a *true* (-1) value, which is also 65655 and beyond the index value needed by most programs. This value is then passed on to a regular jump table (described above). The only requirement is that the jump table recognize *true* as an invalid index value.

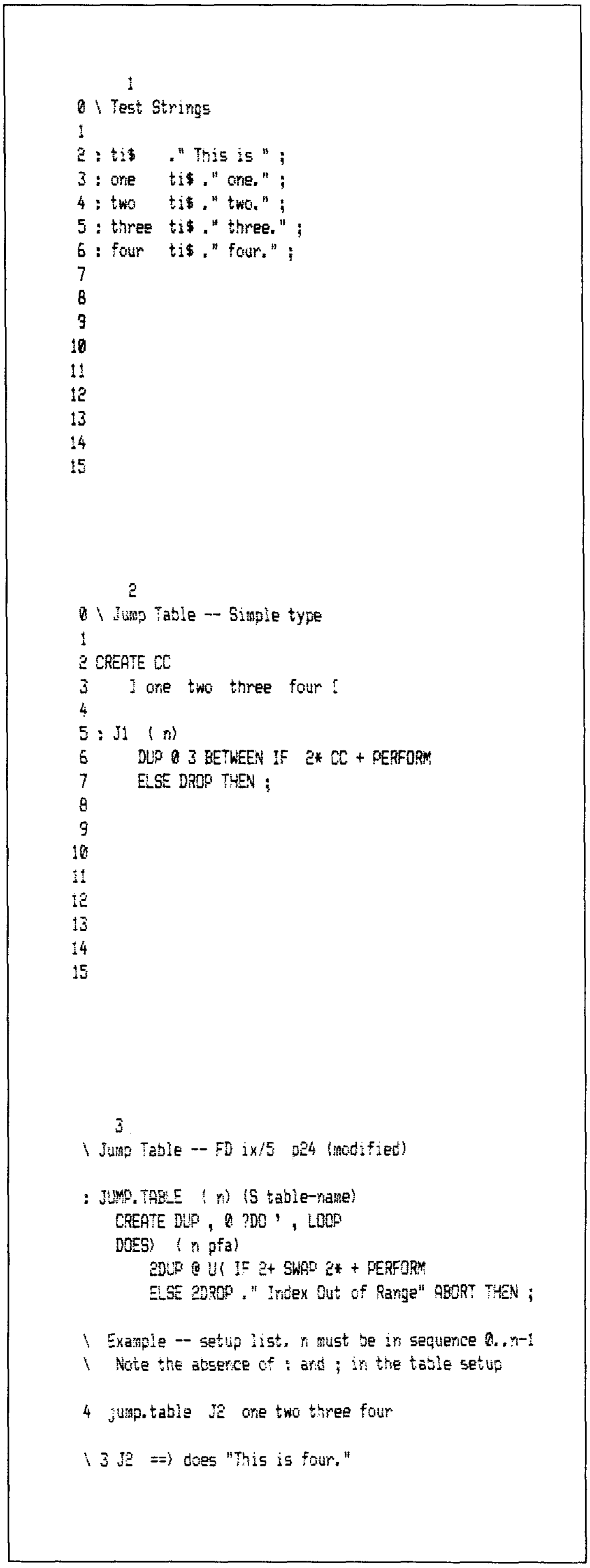
SELTABLE needs to scan a number list and compile it into its parameter field. F83 doesn't have a pre-defined word for this, but you can design your own. The stack holds the count of the number list for the DO LOOP. BL WORD scans for the next numeric string in the data stream. NUMBER converts it into a binary double integer (an error aborts) and puts it on the stack. The DROP converts it into a single integer and , (comma) compiles it. This is a handy way to compile a numeric list without requiring a command after each number.

Unless one of the tables is needed for other purposes, using two tables is clumsy.

In screen six, I show an example by Haskell and McKewan that combines the two tables into a two-dimensional array, and uses one word to handle the job. This word scans a numeric list that is terminated by -1, which also marks the default word. There must be a default word, even if it is NOOP. The 1+ before the WHILE prevents the loop from terminating on a zero, but exits it when a -1 is incremented to zero. It keeps the count for you and stores it at the head of the list.

When the defined table word is run, the selector count indexes a loop to scan the selector column. If a match is found, the pointer is adjusted to the word address. This is then fetched and executed. Otherwise, the default word is performed.

Screen seven shows EXEC.TABLE2, another execution table from Haskell and McKewan. It accomplishes the same task as EXEC.TABLE1, but in a novel way. The three words—EXEC.TABLE, |, and DEFAULT:—are used together but work independently. It's a clever method to solve a complex problem by breaking it up into parts, each



```
     1
\ Test Strings

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

```
      2
\ Jump Table -- Simple type

CREATE CC
    ] one  two  three  four [

: J1  ( n)
     DUP 0 3 BETWEEN IF  2* CC + PERFORM
     ELSE DROP THEN ;
```

```
   3
\ Jump Table -- FD ix/5  p24 (modified)

: JUMP.TABLE  ( n) (S table-name)
    CREATE DUP , 0 ?DO ' , LOOP
    DOES)  ( n pfa)
        2DUP @ U( IF 2+ SWAP 2* + PERFORM
        ELSE 2DROP ." Index Out of Range" ABORT THEN ;

\  Example -- setup list. n must be in sequence 0..n-1
\   Note the absence of : and ; in the table setup

4  jump.table  J2  one two three four

\ 3 J2  ==) does "This is four."
```

```
    4
\ CASE: Jump Table
( Subscripts start from 0 )

: OUT  ( # apf) \ report out of range error & abort
   CR  ." Index out of range on "  DUP BODY> >NAME
   .ID  ."  Max is " @ 1- U. ." -- tried " U. QUIT ;

: MAP  ( # apf - a) \ convert subscript # to address a
   2DUP @ U< IF 2+ SWAP 2* +  ELSE OUT THEN ;

: CASE:  ( n) (S tablename) \ n= number of functions in list
   CONSTANT  HIDE ]  DOES> ( #subscript) MAP  PERFORM ;

4 case: J3   one two three four ;
\ 3  J3 ==> "This is four"
```

```
    5
\ 2 Table Exec. Table
: SEL.TABLE  ( n) (S tablename )
    DUP CONSTANT 0 ?DO
    BL WORD NUMBER DROP , LOOP
    DOES>  ( n pfa - n) \ n= true= no match
    TRUE -ROT DUP 2+ SWAP @  0 DO
        2DUP @ = IF ROT DROP I -ROT LEAVE THEN
    2+ LOOP 2DROP ;

 \ Use as
4 SEL.TABLE S2  4 1 3 2
4 case: J4   four one three two ;
: E1  ( n)  S2  J4 ;
 \ n= index for Jump Table.
 \ Note: n= true (-1) if invalid selection. Jump Table must
 \  detect true as an out of range index value.
```

```
    6
\ Table Exec. Table

: EXEC.TABLE1  (S tablename )
   CREATE HERE 0 , 0
   BEGIN BL WORD NUMBER DROP DUP 1+
   WHILE , ' , 1+
   REPEAT DROP ' , SWAP !
   DOES>  ( r pfa)
       DUP 2+ SWAP @ 0 DO
           2DUP @ = IF 2+ LEAVE THEN
       4 + LOOP  NIP  PERFORM ;

exec.table1  e2  3 three  2 two  4 four  1 one  -1 beep
\ Note: the last Word in list is the default Word, and must be
\   present. It is marked by a -1 select val. Use NOOP if no
\   default is wanted.
```

(Code continues on next page.)

processed by a separate word. (I made a minor bug fix.)

EXEC.TABLE is a defining word. It first sets up the table name, puts the address of the parameter field on the stack, and then inserts a marker space to hold the count of selector values in the list.

| uses the words , and ' to compile the selector values and the word address. It then increments the selector count. The result is a list in which the selector values alternate with the word addresses. Because the code stream between the previous word and | goes through the interpreter, you can use an expression that generates the number placed on the stack. You can write expressions such as CONTROL M or ASCII W instead of just the numeric equivalent as required by EXEC.TABLE1. For example:
CONTROL T | DELWORD

At the end, DEFAULT: drops the duplicated selector count address, then compiles the word after DEFAULT:. Only one word can be compiled. If there is no specific default routine, use DEFAULT: NOOP.

These words define the execution table, but are not a part of it when it's completed. So be careful not to use : (colon) and ; (semicolon) when setting up the table.

You might wonder why a standard, two-dimensional array (2ARRAY) isn't considered, although it can be used. Accessing each value in a 2ARRAY requires the same calculations as for a jump table. This is fine for one item, but is slow for scanning a list. In essence, these execution tables are specially designed for running speed and compiling convenience. Be eliminating EXECUTE or replacing PERFORM with @, you have a data-lookup table, allowing the code to perform double duty.

Wrapping Up

I've described several forms of jump and execute tables. Each takes no more than a screen of code to implement, and can be easily modified to meet program requirements. Examples of use are included in the screens. Although only a single word can be chosen to execute, these tables can simplify the design of large code branches and should be considered as useful alternatives to the ubiquitous CASE and switch statements.

References

Richard E. Haskell and Andrew McKewan, "Vectored Execution and an F83 Full-Screen Editor, *Forth Dimensions* IX/2.

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

```
     7
\ Execution Table -- FD ix/5  p24

: EXEC.TABLE2  ( - a) (S tablename )
    CREATE HERE 0 ,
    DOES>  ( n pfa)
    DUP 2+ SWAP @ 0 DO
        2DUP @ = IF 2+ LEAVE THEN
    4 + LOOP NIP PERFORM ;

: |  ( a n - a)  , ' , 1 OVER +! ;

: DEFAULT:  ( a)  DROP ' , ;

\s If no default function, use as DEFAULT: NOOP
   Example next screen


     8
\ Execution Table -- FD ix/5  p24

 exec.table2  E3
    1 | one    3 | three
    2 | two    4 | four
    default: beep

\   default: noop   \ alternate if no default function.

\ use as 3 E3 -->  "This is three."






```

Forth in Estonia:
A Bit of History

Jaanus Pöial
Tartu University, Estonia

The following overview tries to give an introduction of the place and the history of Forth in Estonia and Tartu.

In Tartu University, there are long traditions in the field of compiler compilers. In the 1970s, professor Mati Tombak started with the parser construction on the IBM/360 computers (the so-called WIRTH-system). Previously, such a system had been implemented for a second-generation Russian computer. This work provided good experience in parsing, but the implementation of language semantics remained primitive enough. Some Estonian scientists began using the attribute method of D. Knuth. Others began looking for an intermediate language for code generation/interpretation.

In 1982, we got acquainted with fig-Forth for the Apple II and discovered that it was just what we needed. There were three of us at that time—Mati Tombak, Viljo Soo, and Jaanus Pöial (the author of this paper). Viljo and I were Mati's students, and graduated from the university as mathematicians in 1982. We started with a new compiler compiler project called TARTU. Mati invented a method to translate from the initial language into Forth (a special kind of syntax-directed translation based on bottom-up parsing). Viljo investigated Forth internals to understand its possibilities for our needs (an excellent FIG model by W. Ragsdale has been our guide for a long time). My research treated the methods of implementing semantics and context checking for compilers via Forth. We decided to write the TARTU system in Forth, because we were tired of rewriting large programs for new and upcoming machines (micros had just become generally available in Estonia).

Our first computer was the Apple II with its 64K. It took about eight months to write the first version of the compiler compiler, TARTU. The CONSTRUCTOR program processed and transformed the context-free grammar of an initial language (including the so-called translation rules) and built tables for the PARSER program. (1 | 1)-MSP method (a mixed strategy of precedence with simple context) was used in this version. The PARSER program translated an initial program text into Forth. Context checking and semantics were written in Forth.

As an exercise, I implemented a language with very unusual control structures—Triodic (N. Goller). It was a challenging order for me from Tallinn Technical University, where mainly the attribute method was used (ELMA system). I included the block structure, dynamic arrays, etc., in this language and finished after two or three months.

In 1984, we moved to a PDP-11. Reino Väinaste and Aivar Juurik (students of economics at the time) wrote their own fig-Forth implementation for this machine and contributed to our group. Viljo started working with a new method of parsing (original and more powerful). Then, the most impressive experience was to feel that Forth is really very portable. An interesting side-effect occurred. When transferring our system, we found a lot of bugs in the Forth kernel (including the model). We now use our system as a test of Forth. Reino also wrote a nice tracer-decompiler for Forth.

The next period of our activities was connected with the Forth circles in the Soviet Union. Several meetings were held (Tallinn, April 1984; Miass, February 1985; Leningrad, October 1985; Tartu, May 1986). We received an order for writing a Fortran-IV translator for quite a strange and obsolete Russian computer with different types of memory (32K + 32K + 64K) and different sets of commands for each type of memory. The Forth system for this machine was written in Leningrad. Unfortunately, it did not meet any standards. It was our great mistake not to start from our own Forth implementation (later, we had to do it anyway). We learned much about separating headers, code, and data. Viljo and Aivar had to master the machine language and write different sets of "colon," "semicolon," "compile," "store," "fetch," etc. This experience was later used to solve the problems of cross-compiling. Now we know that a Forth group must always have a machine/system expert.

Another hard problem was the Fortran syntax (no reserved keywords, spaces allowed anywhere). Viljo (again) invented a parsing method with backtracking. Mati wrote procedure calling, standard procedures, separate compilation and linking for Fortran. Aivar and Reino implemented an I/O and format interpreter, and I processed the declarations of the program. The result was a Fortran translator with about 7K of free space for user

(Continued on page 20.)

Algebraic Specifications of Stack Effects

Jaanus Pöial
Tartu University, Estonia

The most important quality of the Forth word is its stack effect. Particularly strong discipline is required when a large application (hundreds of screens) is written, or when more than two programmers participate in a project. There are some good tools to trace the program, but in a complicated environment, it is an inconvenient task to trace all the program's branches.

The main idea of this work is to introduce a formalism which allows one to check the stack effects according to the program text. The same formalism will be used in the case of Forth programs being generated by some formal mechanism (we will deal with the syntax-directed translation scheme). Our attention is concentrated only on the aspect of parameter passing through the stack, excluding memory handling, I/O, etc.

Each Forth word has an informal specification of its stack effect, given in the form:
input parameter types --- output parameter types

The type lists are ordered, the end of the list corresponds to the top of the stack. This specification does not say anything about the essence of the operation.

Our further investigation is based on the theory of semigroups. In [NP70], M. Nivat and J.F. Perrot introduced a 0-bisimple inverse semigroup called polycyclic monoid. We need some notations to express the main ideas:

A	an alphabet (finite set of type names)
A^*	the set of strings over A (set of type lists)
Λ	the empty string ($\Lambda \in A^*$ for arbitrary A)
ab	the concatenation of strings *a* and *b*
0	the *null specification* (specifies the error situation)

The *set of specifications* over A is the union:
$\Phi(A) = (A^* \times A^*) \cup \{ 0 \}$.

Let $[\, s_1 \text{ --- } s_2 \,]$ denote a pair $(s_1, s_2) \in A^* \times A^*$.

Here s_1 is the list of input parameters and s_2 is the list of output parameters as above. If there is no need to emphasize the alphabet, we use Φ instead of $\Phi(A)$.

The pair $(\Lambda,\Lambda) = [\text{ --- }]$ is called the *empty specification* and is denoted 1.

We may define the *product of specifications* as follows:

1. $\forall\ s \in \Phi : s0 = 0s = 0,$
2. $\forall\ s, t \in \Phi \setminus \{ 0 \} : st = [\, s_1 \text{ --- } s_2 \,][\, t_1 \text{ --- } t_2 \,] =$

$$= \begin{cases} [\, as_1 \text{ --- } t_2 \,]\ , \text{ if } t_1 = as_2\ , \\ [\, s_1 \text{ --- } bt_2 \,]\ , \text{ if } s_2 = bt_1\ , \\ 0\ , \text{ otherwise.} \end{cases}$$

The set Φ is isomorphic to the polycyclic monoid (proof in [NP70]). Consequently:

1. $\forall\ s, t \in \Phi : st \in \Phi,$
2. $\forall\ r, s, t \in \Phi : (rs)t = r(st),$
3. $\forall\ s \in \Phi : s1 = 1s = s,$
4. $\forall\ s \in \Phi : s0 = 0s = 0.$

Let Δ be a set of considered operations. Δ^* is a set of sequences from these operations (set of programs).

Specifications are given by the mapping $s : \Delta^* \to \Phi$:

1. $\forall\ \Pi \in \Delta : s(\Pi) \in \Phi \setminus \{ 0 \}$ is a given specification of the operation Π,
2. $s(\Lambda) = 1$ (the empty program),
3. $\forall\ \omega \in \Delta^*, \forall\ \Pi \in \Delta : s(\omega\Pi) = s(\omega)s(\Pi).$

The program $\omega \in \Delta^*$ is said to be *correct* if $s(\omega) \neq 0$, and *closed* if $s(\omega) = 1$.

We may define a set of correct programs as:
$\text{CORRECT}(\Delta, s) = \{ \omega \in \Delta^* \mid s(\omega) \neq 0 \}$

and a set of closed programs as:
$\text{CLOSED}(\Delta, s) = \{ \omega \in \Delta^* \mid s(\omega) = 1 \}.$

Obviously,

$\{ \Lambda \} \subset \text{CLOSED} \subset \text{CORRECT} \subset \Delta^*.$

These sets are algorithmically solvable because we may calculate the formal specification of a program according to the specifications of existing words. The control structures of Forth need special treatment when writing a practical correctness-checker (an attempt is made to include the correctness-checking into the editor immediately).

Let $s \in \Phi$. The *inverse element* $s^{-1} \in \Phi$ is defined by the

conditions:
1. if $s = 0$, then $s^{-1} = 0$,
2. if $s = [\ s_1 \text{ --- } s_2\ ]$, then $s^{-1} = [\ s_2 \text{ --- } s_1\ ]$.

The *partial order relation* ≤ is convenient in the theory of semigroups ([CP67]): $s \leq t$, iff $st^{-1} = ss^{-1}$. Since $0t^{-1} = 00 = 0$, we have $0 \leq t$ for all $t \in \Phi$.

Theorem One. The following assertions are equivalent:
1. $[\ s_1 \text{ --- } s_2\ ] \leq [\ t_1 \text{ --- } t_2\ ]$,
2. $\exists\ a \in A^* : [\ s_1 \text{ --- } s_2\ ] = [\ at_1 \text{ --- } at_2\ ]$,
3. $[\text{ --- } s_1\ ][\ t_1 \text{ --- } t_2\ ][\ s_2 \text{ --- }\ ] = 1$,
4. $[\text{ --- } s_1\ ][\ t_1 \text{ --- } t_2\ ] = [\text{ --- } s_2\ ]$,
5. $[\ t_1 \text{ --- } t_2\ ][\ s_2 \text{ --- }\ ] = [\ s_1 \text{ --- }\ ]$.

Having a partial order relation, the problem of comparable elements arises. At present, we know that the null element is comparable with all elements of Φ.

Theorem Two. The following comparability conditions are equivalent for the elements of Φ:
1. $s \neq 0$, $t \neq 0$ and s is comparable with t,
2. there exists an element $r \neq 0$ so that $r \leq s$ and $r \leq t$,
3. there exists an element $u \in \Phi$ so that $s \leq u$, $t \leq u$, and at least one of the conditions $st^{-1} \neq 0$, $s^{-1}t \neq 0$ holds.

Further, we need a method to solve inequalities given by such a partial order relation. These inequalities may have a "recurrent" form like $s \leq rst$.

Theorem Three. Inequality $s \leq rst$ by $s \neq 0$ holds in Φ, iff there exist $a, b, c \in A^*$ so that
$ar_1 = s_1$, $ar_2 = bs_1$, $ct_1 = bs_2$, and $ct_2 = s_2$.

We finish the study of algebraic properties of Φ with observing infimum and supremum of subsets of Φ.

An arbitrary two-element subset $\{\ s, t\ \} \subset \Phi$ has the greatest lower bound, which may be expressed as

$$\inf\{\ s, t\ \} = \begin{cases} s, \text{ if } s \leq t, \\ t, \text{ if } t \leq s, \\ 0, \text{ if } s \text{ and } t \text{ are non-comparable.} \end{cases}$$

This definition is obvious (see also Theorem Two). The notion of supremum is more complicated. For the null element, we may define $\sup\{\ r, 0\ \} = r$ in the case of all $r \in \Phi$. Let $s, t \in \Phi \setminus \{\ 0\ \}$. If there exist $a, b, c, d, e \in A^*$ so that $s = [\ abd \text{ --- } abe\ ]$, $t = [\ cbd \text{ --- } cbe\ ]$, and the length of b is chosen maximal (possible), then there exists
$\sup\{\ s, t\ \} = [\ bd \text{ --- } be\ ]$.

This choice of b guarantees the defined upper bound to be the least (see also Theorem One). If it is impossible to choose these five strings in any way, then no supremum exists.

A set of stack operations Δ and the homomorphism $s : \Delta^* \rightarrow \Phi$ induce two languages, named CORRECT(Δ, s) and CLOSED(Δ, s) before. A program $\omega \in$ CLOSED(Δ, s) as a whole has neither input nor output parameters. At the same time, parameter types inside of ω are compatible, i.e., ω is correct. All "user-oriented" programs must be closed, because the stack is only an implementation-level tool. This point of view evokes our special interest in the closed programs.

We investigate the syntax-directed translation scheme ([AU72]) and try to answer the question if there exists an algorithm for detecting whether or not a given scheme generates only closed programs.

The *syntax-directed translation scheme* is a quintuple T = (N, Σ, Δ, R, S), which consists of the following components:

N a non-terminal alphabet,
S ∈ N a fixed initial symbol (an axiom),
Σ an input alphabet,
Δ an output alphabet, and
R a finite set of translation rules of the form:

$A_0 \rightarrow x_0A_1x_1 \ldots x_{n-1}A_nx_n\ ,\ z_0B_1z_1 \ldots z_{n-1}B_nz_n$
($x_i \in \Sigma^*$, $z_i \in \Delta^*$, A_i , $B_i \in N$), by which the vector (B_1, ..., B_n) is some permutation of the vector (A_1, ..., A_n).

If (B_1, ..., B_n) = (A_1, ..., A_n) for all rules of R, then the syntax-directed translation scheme is said to be *simple.*

The syntax-directed translation scheme defines a set of pairs $(\sigma, \omega) \in \Sigma^* \times \Delta^*$, which may be derived from the pair (S, S). The first components of these pairs constitute an input language of the scheme; the second components constitute the output language. The string ω is said to be the translation of the string σ. The translation scheme may also be treated as a pair of grammars T = (G_1, G_2), defined by R.

Let the input grammar G_1 be a reduced, context-free grammar ([AU72]). For the output grammar, we use a notation G_2 = (N, Δ, P, S). The output language of T is a set

$$L_2 = \{\ t \mid t \in \Delta^* \ \&\ S \underset{G_2}{\Rightarrow^+} t\ \}.$$

Let the output symbols $\Pi \in \Delta$ have specifications s(Π), i.e., $s(\Delta) \subset \Phi \setminus \{\ 0\ \}$.

The syntax-directed translation scheme T is said to be *correct* if $L_2 \subset$ CLOSED(Δ, s).

The system of inequalities I (T, s) is defined:
1. An unknown $Z(A) \in \Phi$ is introduced for each $A \in N$.
2. The rules of G_2 are replaced by inequalities—the rule of form $A \rightarrow X_1 \ldots X_k$ induces $Z(A) \leq Y_1 \ldots Y_k$, where $Y_i = s(X_i)$, if $X_i \in \Delta$, and $Y_i = Z(X_i)$, if $X_i \in N$. If the right part of the rule is empty, then we take $Z(A) \leq 1$.
3. The inequality $1 \leq Z(S)$ is added where S is the axiom of the scheme T.

The following auxiliary sets are introduced for each nonterminal symbol $A \in N$:

$C(A) = \{ (u, v) \in \Delta^* \times \Delta^* \mid S \Rightarrow^* uAv \},$
$L(A) = \{ \omega \in \Delta^* \mid A \Rightarrow^+ \omega \}.$

Theorem Four. The following assertions are equivalent:

1. the syntax-directed translation scheme T is correct,
2. the system of inequalities I (T, s) is solvable,
3. for each nonterminal symbol $A \in N$ there exists a supremum
   $m(A) = \sup \{ [s(vu)]^{-1} \mid (u, v) \in C(A) \},$

   by which the following inequality holds
   $m(A) \leq \inf \{ s(\omega) \mid \omega \in L(A) \}.$

This theorem allows one to check an initial translation scheme for which Forth is the output language.

It may happen that it is hard to classify the parameters of stack operations because there are many type-independent operations like DUP, SWAP, and DROP, etc. in Forth. In such cases, it is useful to introduce "wild card" (or "free") symbols which are able to replace an arbitrary type name (let us use asterisks to express "wild card" symbols). The following examples of specifications are used to illustrate this approach:

DUP	[* --- * *]	Copies the top element on the top of the stack.
SWAP	[** * --- * **]	Interchanges the two top elements.
!	[* *addr* ---]	Stores the element of arbitrary type at *addr* (mixed specification).

It is possible to generalize the operation of multiplication for "wild card" symbols with some restrictions (no new "wild cards" may appear on the right side of any specification).

References

[AU72] Aho A.V., Ullman J.D. *The Theory of Parsing, Translation and Compiling*, Volume 1: Parsing. Englewood Cliffs, NJ, 1972.

[CP67] Clifford A.H., Preston G.B. *The Algebraic Theory of Semigroups*, Volume 2. Rhode Island, 1967.

[NP70] Nivat M., Perrot J.F. Une Généralisation du Monoïde Bicyclique. *C.R. Acad. Sci. Paris*, 271A, 1970, pp. 824–827.

Jaanus Põial is an Associate Professor of theoretical computer science at Tartu University, where he also has served as head of the Department of Computer Science. He may be reached at his jaanus@cs.ut.ee e-mail address.

(Bit of History, continued.)

programs (Forth code) and 64K for data. It worked better than BASIC for the same machine but, of course, could not find a real application.

We had a fine team at that time which was able to solve difficult problems at any moment. Viljo finished his research with a (1,1)-DMSP parsing method. I wrote my Ph.D. in 1986 on the topic of formal specifications of Forth programs (I used algebraic methods to describe formally the stack effects of Forth words). Mati started with the cross-compiling problems. Unfortunately, Reino and Aivar left their jobs at the university. Professor Ain Isotamm joined us after a long period (about two years) of "ripening." He was, and still is, a true programming ace, whose relations to Forth and Forthers were [at the time] friendly but indifferent. Now he is a real Forth enthusiast who has understood that Forth is a philosophy of programming, not [just] "one more language."

The last period in the history of our group began with the IBM PC clones and the Forth-83 Standard. We translated all our programs from fig-Forth into Forth-83. Most of this work was quite formal, but some algorithms with the LEAVE operator had to be revisited. It became clear that 32-bit computers were coming, and we started with a new Forth project, the 32-bit Forth-83/32. There were some difficulties in overcoming word-length problems. We have virtual 32-bit addresses in Forth-83/32 (unsigned arithmetic works on the addresses). Packing and unpacking real addresses makes the system slow. Regardless of this aspect, we found it to be useful to work "for the future." The main authors of Forth-83/32 are Reino and Aivar (we were able to engage them once more as experienced Forthers). Viljo has now begun taking care of the system. The true life of Forth-83/32 begins on a real 32-bit architecture.

The first big project, which uses Forth-83/32 and TARTU, is a Modula-2 translator and cross-compiler. The Modula-2 translator is nearly finished. Ain is writing a database system which rests on the data model (and implementation) of Modula-2 (screen input/editing, table-format output, etc.). We have always had some students working with our group. Toomas Saarsen is one of the most prominent young Forthers now. He wrote a compiler which generates machine code from the Forth environment (not from the text, but from threaded code). It quickened our parser by four to five times.

This is the story of our group. In Estonia, there are more Forth groups, mainly in Tallinn (Estonia Radio, Tallinn Technical University, Institute of Cybernetics). On the whole, Forth is not very popular or well supported in Estonia; however, we keep doing our work and moving on. Unfortunately, we do not have any links with the world-wide Forth community. We are interested in all Forth-related events and projects (standardization processes, Forth education projects, etc.). We cherish the hope that this isolation will be broken.

Forths in the Design, Test & Extension of an HDTV Format Convertor

Philip S. Crosby
Beaverton, Oregon

In about 14 months (of 80-hour weeks) starting in February 1990, three full-time-equivalent engineers (two full-time, three part-time) built an HDTV Format Convertor for the Advanced Television Test Center to be used for the generation and evaluation of video in the four proposed U.S. HDTV standards. Comprising a total of about 300 ICs, and having A/D and D/A conversion process quality suitable for production of high-quality video to be viewed by expert observers, most of our efforts were concentrated on analog and digital hardware development.

Our use of Forth was extensive, from the early hardware feasibility demonstrations and clock-accurate simulations, to the generation of target code for the 8051 microcontroller, and the test and calibration routines used in production. Many dialects were used, including F83, a83 (a 32-bit, public-domain Amiga F83 work-alike), and Bryte-Forth (a fig-Forth-like dialect for the 8051). Finally, we ported the code generator and simulator to F-PC to simplify product support and future extensions. The bulk of the Forth programming was done by the two full-time HW engineers. The finished application compiled to about 7.5 K, of which about 2.7 K was data for the HW state machines.

...the accounting over a 16.67 mSec video field involves almost 70 million common time units.

The Chicken and the Egg

A problem that is nearly inherent in the testing of new television scanning standards is the inability to record video sources and the results of video processing operations in the standard under evaluation. Video tape recorders, including digital recorders, are extremely standard-specific and are terribly costly to design. Consequently, when four different scanning standards were proposed to the FCC, it was feared that the lack of a video recording means would cast doubt on the outcome of any kind of testing process.

Fortunately, a digital video tape recorder (DVTR) did exist for one HDTV standard, the 1125-line standard (1920w x 1035h) developed in Japan in the 1980s. The Advanced Television Test Center (ATTC), charged with the responsibility of conducting the tests, approached Tektronix about the possibility of building hardware that could accept the analog signal from the proponent, digitize it, and format the signal as a bitstream that would "fool" the DVTR into believing that the signal was in the digital 1125-line format. Upon playback, filler codes would be removed and the original line and field formats would be regenerated, resulting in an analog signal that would be nearly indistinguishable from the original.

Of course, all of this had to be done in as little time as possible and the performance of the device, dubbed a Format Convertor (FC), had to be such that the signal quality was equal to that of the $400,000 DVTR. But, if the FC could be built, it would end the "chicken and egg" problem, enabling various HDTV formats to be tested on a level playing field.

The Design Fundamentals

In order to give us power and room to work in, and to provide a rack-mountable mechanical package, we chose the Tektronix VX1505 mainframe to house the FC modules. The VXI (Vme eXtensions for Instrumentation) D-size module offers extensive slot-to-slot interconnect at a 50 ohm impedance level, ideal for handling the 16-bit video (eight bits luminance and eight bits chrominance) and the various timing and qualification signals needed.

However, we saw no need for the complexity of the VME bus that was on the passive backplane. Instead, we used 11 lines of the P2 VME bus for our 8051 microcontroller running Bryte-Forth, a dialect of fig-Forth that had been used in the lab and had worked out very nicely for prior turnkey projects. The 8K kernel supports an auto-bauding RS-232 interface. Although Bryte-Forth assumed that bubble memory would be used for mass storage (remember bell-bottoms, disco?), we had no need for mass storage.

The A/D Module

We start with an analog RGB signal and a sync signal, either H and V drive or composite sync. The horizontal sync component is used to phaselock the user sampling clock at a programmed multiple of the line rate, around 75 MHz. The analog RBG signals are matrixed to luminance and color difference components and are digitized, using the top eight bits of three ten-bit A/D converters. The samples from the A/Ds are muxed with samples generated by a proprietary Zone Plate Generator (ZPG) chip, used for calibration and diagnostics. A reset pulse and the frequency divided user clock are sent to the next module, along with the active video samples and the associated clock and write gate.

The FIFO Modules

The buffer memory between the user video and the output from the FC to the DVTR uses 1:4 de-muxed IDT FIFO chips for the Y and C channels. The FIFO organization keeps things simple and expandable, and places no requirements on the controller bus. (One engineer was available for only a short time at the beginning of the project—we had him do the FIFO board to define the signal interfaces before anything else was done.)

To ensure that all of the modules work properly when mounted on 400 mm board extenders, the FIFO read protocol is a bit complex. The FIFO module outputs two different data signals in response to two different read gates, a user read gate and a dummy read gate. When the dummy read gate is asserted, the FIFO outputs a delayed clock, a delayed read gate, and filler samples (byte constants that describe "impossible" colors). Assertion of the user read gate results in a stream of user samples read from the FIFO. Since a continuous clock and the qualifying gate travel in the direction of data flow, time delay due to a board extender cannot skew the timing relationships. Identical FIFO modules are employed for recording and playback.

The I/O Module

The I/O module employs a PLL multiplier to generate the DVTR sample clock from the frequency divided user clock. It then reads the user data from the record FIFO, formats it to look like a digital 1125-line signal to "fool" the DVTR into accepting the signal.

On playback, timing information is extracted from the signal from the DVTR, filler samples are detected and removed (and those in the first active line of DVTR field 1 are counted to allow automatic detection of the playback standard), and the user samples are written to the playback FIFO. The DVTR sample clock is divided and sent through the playback FIFO to the D/A module.

The D/A Module

The user clock is generated from the divided DVTR clock using a PLL multiplier again. User-specific sync and blanking signals are generated. The signal is D/A converted using proprietary D/A chips and matrixed back to RBG.

General Constraints

To reduce crosstalk, all signal interfaces and signal clocks are at ECL levels. Clocks and gates are conveyed between modules differentially. Variations in clock arrival times are held to a few tens of picoseconds. (One least significant bit of an eight-bit 30 MHz sinewave is traversed in about 50 pSec!)

The FC was designed to be as "soft" as possible (but no softer). The hardware was designed to allow ease of programming. Although spreadsheet simulations had indicated that 2K sample signal FIFOs would suffice, we would need to do further simulation to determine just how much buffer preload would be required.

How Forth Contributed to the Format Convertor

The First Behavioral Simulation

The user rate had to differ from that of the DVTR so that an integer number of samples would be taken per user line. The exact frequency ratio is 56 user samples per 55 DVTR samples, corresponding to sample frequencies of about 75.52 MHz and 74.176 MHz, respectively.

Keeping track of the read and write buffer transactions require accounting for the number of common time units (CTUs), time slices that are the greatest common divisor of the two sample periods. One CTU is about 240 pSec. Thus, the accounting over a 16.67 mSec video field involves almost 70 million CTUs.

Fortunately, one of my home machines is an Amiga. A83, a 32-bit public-domain version of F83 enabled writing the simulations in a couple of days. Furthermore, the Amiga's multitasking capabilities let me run a new simulation while editing the previous 50-page RAM file in EMACS to pick out the interesting parts of the simulation.

The H and V State Machines

Signal timing events on the A/D, I/O, and D/A modules are generated by an inner (H) loop clocked at the signal sampling rate and an outer (V) loop clocked by one bit derived from the inner loop. The inner loop usually subtends one line, while the outer loop subtends a frame.

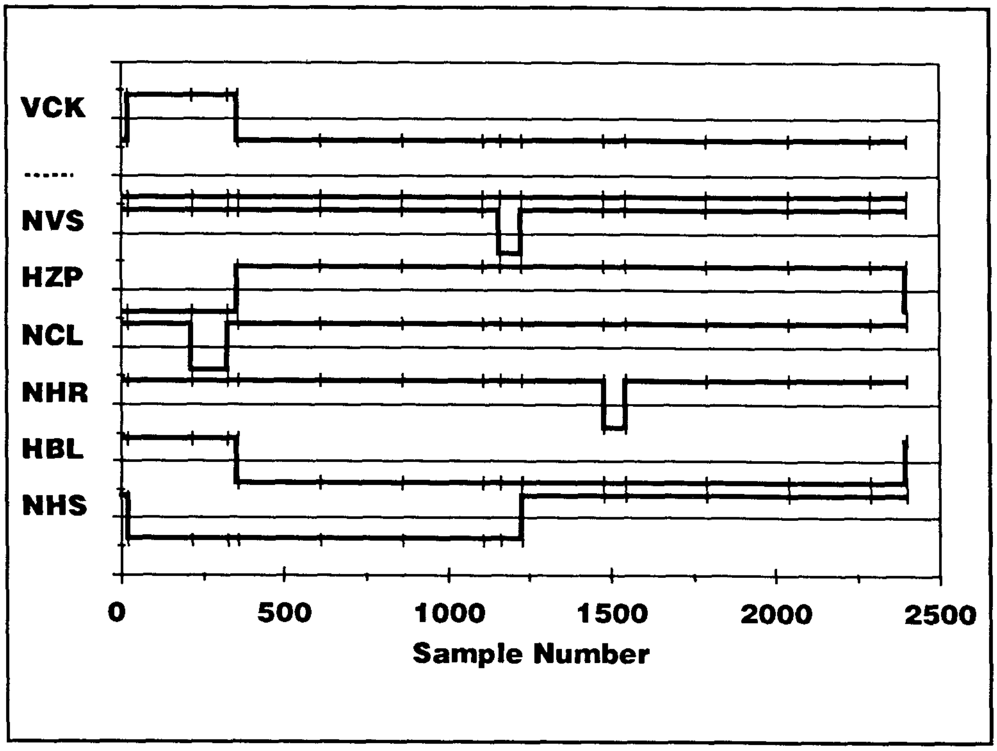
The controller sees a control port and a data port for each state machine. Writing a zero to the control port resets the state machine memory, which is a FIFO chip. The controller then writes interleaved data and counter preload bytes to the FIFO.

The state machine only reads the FIFO. However, a PAL asserts the FIFO chip's retransmit bit when its empty bit goes true, creating a loop. In this way, the state machine latches the data byte for the number of clocks determined by the associated preload byte. The set of data-count pairs determines the bit patterns generated by the state machine. Note that the state machine has no explicit loop counter, simplifying its design. The downside of this design for the H state machines is that there is a minimum interval that can be specified, due to the limited speed of the FIFO and the interleaving of data and count bytes. For the state machines used in the FC, the overhead is 11 clocks, about 150 nSec. However, although we need to program specific events to the nearest sample, we do not expect to have to space events by less than a few hundred nSec. One bit of the H state machine is used to clock the V state machine, providing the programmer with control over the H-to-V "phase" relationship. The V state machine is clocked once per line in the A/D and I/O modules, but, due to the need to generate timing details for interlaced standards, the V state machine in the D/A module is clocked twice per line in most standards.

To ensure proper relative phasing of the state machines, an index signal and a clock at the greatest common divisor of the user and DVTR sample frequencies is passed along with the video samples. The index signal also serves to initialize the record and playback FIFO modules. The leading edge of the index signal forces the same retransmit condition to occur in the state machines as normally occurs when the state machine FIFOs are empty.

Figure One depicts a typical state machine output. The sample clock is 75.52 MHz. The most important signals are NHS, the square wave whose falling edge is locked to incoming H sync, HBL which inhibits the writing of signal samples to the record FIFO and NHR, which establishes the sample number at which the following state machine (in the I/O module) is to be reset. NHR is combined with an NVR bit from the vertical state machine to specify the exact point in the user frame(s) that indexes the I/O state machine. The rise of VCK clocks the A/D's V state machine.

Figure One. A/D H state machine bit patterns.



When specifying the data bits and the times that they become effective, it is convenient for the programmer to specify the times relative to the format of the signal being handled. The I/O state machine, for example, is defining how the signal going to the DVTR is formatted. While it is indexed at a time that precedes the start of active video by about 2/3 line, all other events that the programmer is concerned with relate to the signal for the DVTR. So, we wrote a code generator to simplify the programming of the state machines.

All timing is referenced to the nominal start of blanking of the signal handled by the state machine. A VAR (same as F-PC's VALUE) is assigned a value by the word INTO. The code generator was written in F-83. Representative input to the code generator appears in Figure Two.

The general parameters (counter overhead, offset to index pulse, and loop duration) are loaded into VARs. The structure is declared, and Cycle.Start sets up a >MARK and puts 0 (item count) on the stack. |b and |d are shorthand for base conversion. If the specified time (TOS to ?,) is less than Offset.Counts, the data and time values remain on the stack and the item count is incremented, else the data and time are compiled. Cycle.End compiles any remaining data-time pairs and resolves the >MARK with the item count.

fifo.correct converts the time values into counter preloads, taking Overhead into account (later revisions replaced Overhead with Term.Count, a more sensible

Figure Two. Sample input to the code generator.

```
\ H Fifo Setup for  525 line  IO -                 05 Apr 91 psc
 11 into Overhead       ( V.NH1122 ) 1051 into Offset.Counts
                        ( C.2BRRRR ) 2200 into  Cycle.Counts
                        ( K.DL2121 )    variable FIFO.Struct
                        ( |||||||| )    Cycle.Start
( HBlank Start      ) |b  10111111  |d      0              ?,
( repeat            ) |b  10111111  |d     202             ?,
( HBlank En         ) |b  00101010  |d     275             ?,
( 1R1 end           ) |b  00101110  |d     280 42 +        ?,
( 2 dummies         ) |b  00001110  |d     522             ?,
( 2 dummies         ) |b  00001110  |d     572             ?,
( 2 dummies         ) |b  00001110  |d     622             ?,
( 2 dummies         ) |b  00001110  |d     772             ?,
( 2 dummies         ) |b  00001110  |d    1022             ?,
( index received    ) |b  00101110  |d Offset.Counts       ?,
( repeat            ) |b  00101110  |d    1272             ?,
( 2R2 st            ) |b  00101100  |d    1340 4 -         ?,
( 2R1 end,          ) |b  00101101  |d  280 1100 + 42 + 10 +  ?,
( 2 dummies         ) |b  00001101  |d    1522             ?,
( 2 dummies         ) |b  00001101  |d    1772             ?,
( repeat            ) |b  00101101  |d    2022             ?,
( 1R2 start         ) |b  00100101  |d  Cycle.Counts 40 -  ?,
 Cycle.End

fifo.struct  dup fifo.correct  fifo$ saveas 525ioh
forget fifo.struct
```

descriptor). `fifo$` converts the parameter field of `Fifo.Struct` into a pair of two-digit hex numbers and a count, as shown below

```
2E  2E
2E  CB
2C  AB
2D  B1
0D  11
0D  11
2D  81
25  E3
BF  41
BF  C2
2A  DC
2E  43
0E  D9
0E  D9
0E  75
0E  11
0E  EE
11
```

Eight such files are generated for each TV standard that the FC operates on. These files are collected by a script file that originally ran on WordStar, resulting in the generation of Bryte-Forth source code (See Figure Three).

The `.FI` directive tells WordStar to insert <filename> into the stream that forms the output file. The words { `A/D | I/O | D/A` } and { `HFIFO | VFIFO | HFIFO1 | HFIFO2` } are combined by `FLOAD` ("`Fifo LOAD`") to form the port address for the state machine memory. `FLOAD` is a `<BUILDS DOES>` word whose run-time definition causes the child word to load the proper state-machine memory. A little manual bit diddling in the final colon definitions takes care of the loose ends.

The Rest of the Bryte-Forth Code

The user interface is fairly simple; there are eight lights and eight LEDs. Four of the buttons select the standard. A fifth button selects the standard selection mode, record, playback, or auto-detected playback. The sixth button, operative only in record mode, selects the input sync source. The seventh button permits bypassing of the recorder for setup or diagnostic purposes. The eighth button enables external RS-232 for external control and for downloading calibration programs.

In addition to controlling the state machines and handling the user interface, the resident code supports operation of an important diagnostic tool, the ZPG. It generates sine samples that are a function of the X, Y, and T (frame number) pixel values, and can be selected to provide the signal samples, rather than the A/D converter chips. Mike Cranford, the engineer responsible for most of the analog and firmware work, got his introduction to Bryte-Forth in writing the interface to the ZPG. He got a nicely decomposed interface running in about two days, and wrote most of the rest of the Bryte-Forth code residing in the FC.

Downloadable diagnostics and calibration code make use of the ZPG and custom test signals from the Tektronix TSG1001 Test Signal generator greatly ease manufacturing. Although a small number of FCs have been built to date, we support our customers with next-day module exchange. Most field problems have been identified in less than an hour.

Follow-On Work

After initial manufacture, we ported all of the FC-related code to F-PC, largely because of the file I/O and extensive on-line help system. The file merging that had been performed by WordStar is now done in F-PC. A clock-accurate simulator running on the output of the code generator is now in use and was needed to handle

Figure Three.

```
.PO 0
.PL 0
.PF OFF
( Start of Data Area for Control Stores - July 2, 1992)
.FI 525ADTH
A/D HFIFO  FLOAD  525A/DTH

.FI 525ADTV
A/D VFIFO  FLOAD  525A/DTV

.FI 525ADDH
A/D HFIFO  FLOAD  525A/DDH

.FI 525ADDV
A/D VFIFO  FLOAD  525A/DDV

.FI 525IOH
IO HFIFO  FLOAD  525IOH

.FI 525IOV
IO VFIFO  FLOAD  525IOV

.FI 525DAH
D/A HFIFO1  FLOAD  525D/AH1

.FI 525DAH
D/A HFIFO2  FLOAD  525D/AH2

.FI 525DAV
D/A VFIFO  FLOAD  525D/AV

: 525REC     02 ADSTAT BSET  80 ADSTAT BCLR
             ADSTAT C@  A/D WRTPORT
             525IOH   525IOV  COMP?
             IF     525A/DTH 525A/DTV  01 ADSTAT BCLR
             ELSE   525A/DDH 525A/DDV  01 ADSTAT BSET
             THEN   ADSTAT C@  A/D WRTPORT ;

: 525PB      40 DASTAT C!
             DASTAT C@  D/A WRTPORT
             525D/AH2 525D/AV ;

BASE @ DECIMAL
56 CONSTANT 525P  BASE !
```

a modification for the European HDTV systems. There, because of the field rate difference, five user frames subtend six DVTR frames, which means that the state machine accounting must cover nearly 10^9 CTUs. The simulator has become a necessity.

The FC could prove to be the foundation for worldwide HDTV program exchange, since a *de facto* standard for a multiformat digital interface has been produced as a result of our efforts.

How Well Did Forth Work?

Very well, thank you. Had we followed the classic project organization, with hardware and software people on opposite sides of the fence, the project could have been a disaster. We had enough problems with specification changes, component problems, and the need to anticipate last-minute format changes that we had been assured "would never happen." It's nice that, contrary to the way things usually work, the software wasn't a problem.

The number of Forth dialects wasn't really a problem, either. The author's Forth experience began with a fig-Forth dialect for the Osborne 1 and has included seven other dialects. The unifying factor, however, is the Forth virtual machine and the fact that Forth, fundamentally, makes sense.

Suggested Improvement

Our use of Bryte-Forth parallels that of many other efforts where RS-232 is the interface between the host and the target. The host runs some sort of terminal program (we used PCPlus) and the Forth source is uploaded to the target machine. However (and perhaps this is an old idea), if the terminal program were context sensitive, it seems that it could assemble CODE definitions in the target machine using an assembler residing in the host. If the target processor is known and the target kernel contains HERE, >MARK, <MARK, >RESOLVE, and <RESOLVE, it should be possible for a smart terminal program to remotely assemble code in the target machine. It would have helped us some in the FC project, and would be very nice in a subsequent application we have in mind.

Acknowledgments

Thanks to Paul Barton, whose technician work on the FC and his advocacy of Bryte-Forth got us going in the right direction; Mike Cranford, who did the bulk of the target code; Jan Kuderna, for her technician work and, later, her F-PC starter project, the file insertion code; and Jim Geddes, for his work on the test and cal routines, and for handling most of the update work.

Thanks also to the developer of Bryte-Forth, whoever he/she may be. We have tried to contact the developer with no success. But most of all, thanks to the Forth Interest Group and the Forth community for their effort in creating and extending a truly "enabling technology."

Philip S. Crosby is a Principal Engineer for Tektronix, Inc. This paper was originally presented at the 1993 FORML Conference.

Forth Interest Group
Statement of Change in Financial Position
April 30, 1993 to April 30, 1994

	4/30/93	4/30/94	Change
ASSETS:			+ = Increase
			- = Decrease
Current Assets:			
Money Market	23,740.48	21,215.29	-2,525.19
Checking & Cash deposits	9,855.61	5,329.16	-4,526.45
Total Current Assets:	33,596.09	26,544.45	-7,051.64
Inventory:			
Inventory at cost	16,280.00	15,681.00	-599.00
Total Inventory:	16,280.00	15,681.00	-599.00
Other Assets:			
Second Class Postal Account	161.10	119.78	-41.32
Accounts Receivable	500.00	0.00	-500.00
Equipment	5,826.02	16,361.22	10,535.20
Total Other Assets:	6,487.12	16,481.00	9,993.88
TOTAL ASSETS:	56,363.21	58,706.45	2,343.24
LIABILITIES:			
Sales Tax	100.88	19.11	-81.77
FD Dues Alloc to future months	29,526.10	30,694.23	1,168.13
TOTAL LIABILITIES:	29,626.98	30,713.34	1,086.36
Financial Reserve:	26,736.23	27,993.11	1,256.88

STUCK ON STACKS

A guide for stacrobats and stacropobes.

Gordon Charlton
Hayes, Middlesex, England

Forth is different to other languages in having a thing called a stack. Of course other languages have a stack, but they are not like the Forth stack. Typically the stack means the return stack, but in Forth it means the data stack. The return stack is called the return stack to differentiate it from the stack, and not to indicate that it may have return addresses on it. Of course it may, but then it may not. You can put things on the return stack, or take them off, as long as they are not return addresses. The data stack, or stack, is for data. It is not for return addresses either. There are lots of different sorts of data that can go on the stack, but all that Forth knows about them is their size, and it only knows that most of the time. Some data may go on the data stack, or then again it may not. Floating-point data, for instance, may go on the data stack, or it may go on the floating-point stack, which is another stack that Forth may or may not have. Because it may or may not have its own stack, Forth programmers have to write programs as if it has its own stack, and as if it does not.

A stack is a very simple data structure that only allows two operations, which are called PUSH and POP. The Forth standard defines fifteen words in the core and core extensions that operate on the data stack alone, none of which are either PUSH or POP. In addition, it defines six words that operate on the return stack. None of those are called PUSH or POP either. Neither is the one defined in the double extensions. It also defines another six that may act on the data stack, but you can't assume they do because 1) they may act on the floating-point stack, and 2) even if they don't, we have no idea how wide a floating-point number is anyway.

Of the stack words mentioned above, there are two you are not supposed to use because they pretend the stack is not a stack, but an array. This array is upside down, in that the top of the stack is the bottom of the array. Using the words PICK and ROLL causes the top of stack, and hence the upside-down array, to move, so they are not a good idea. The compilation stack (another stack that may or may not exist during compilation, and may or may not be the data stack, of which we do not know how wide the datums on it are) has only two operators. They are analogous to PICK and ROLL, and were chosen, presumably, because they are a good idea.

These are not the only stacks that may or may not exist. The return stack may or may not play host to the loop-index stack and the locals stack, which must both operate as if the other one was not in the same place. Of course the programmer must act as if they are on the return stack. As the return stack is used for just about everything except for return addresses, two words are provided to allow return addresses to be removed from the return stack under programmer control. These are CATCH and THROW. They work like this: after you have nested down into a number of subroutines you can THROW up through them again, but remember that if you are going to THROW up you need to CATCH it. Of course THROWing does not only affect the return stack, it also affects the data stack and the floating-point stack (if there is one). Once you have THROWn, the data stack may or may not have items on it that are undefined. So after a CATCH, you have to DROP them.

Finally, we do not know how any of these stacks are implemented. Often they grow downwards in memory, which of course means that the upside-down array is really the right way up, but then again they may have been implemented in a separate memory space. Indeed, the floating-point stack might even be implemented on the stack in the floating-point coprocessor, if you have one, but it is unlikely, as it is unlikely that the floating-point stack in the floating-point coprocessor is much like the floating-point stack defined in the Forth standard. Indeed, as far as the programmer knows, the hardware might very well have a little man inside it writing numbers on plates and pushing them onto the sort of spring-loaded plate dispensers that no-one has ever actually seen in cafeterias, but that abound in books that claim to make the stack simple. Of course in this sort of implementation a hardware crash is more literally true than in the average system, and if you ever have one you could be picking fragments of china off the bus and out of your chips for ages.

Well, if this has helped to clarify any of the confusion about Forth stacks please let me know and I will attempt to correct the situation.

Gordon Charlton was named after Charlton Heston. He has been described as tall, handsome, witty, intelligent, charming, and world famous. Gordon is also quite tall. He wishes to thank readers of *Forth Dimensions* for not responding to his request for a rigorous description of the Ratcliffe-Obershelp algorithm, as this prompted him to develop a rather better one himself He may be reached via e-mail at gordon@charlton.demon.co.uk.

Statement of Ownership, Management, and Circulation

1. Publication Title: *Forth Dimensions*
2. Publication No. 0002-191
3. Filing Date: Oct. 1, 1994
4. Issue Frequency: bi-monthly
5. No. of Issues Published Annually: 6
6. Annual Subscription Price: $40
7. Complete mailing address of known office of publication: 4800 Allendale Avenue, Oakland, California 94619
8. Complete mailing address of headquarters or general business office of Publisher: same as above
9. Publisher: Forth Interest Group, 4800 Allendale Avenue, Oakland, California 94619. Editor: Marlin Ouverson, 5364 East Avenue G, Lancaster, California 93535. Managing Editor: Forth Interest Group, 4800 Allendale Avenue, Oakland, California 94619.
10. Owner: Forth Interest Group (non-profit), 4800 Allendale Avenue, Oakland, California 94619.
11. Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: none.
12. The purpose, function, and nonprofit status of this organization and the exempt status for federal income tax purposes: has not changed during preceding 12 months.
13. Publication Name: *Forth Dimensions*
14. Issue Date for Circulation Data Below: XVI/3

15. Extent and Nature of Circulation:	*Average No. Copies Each Issue During Preceding 12 Months*	*Actual No. Copies of Single Issue Published Nearest to Filing Date*
a. Total No. Copies *(net press run)*	1300	1300
b. Paid and/or Requested Circulation		
(1) Sales through Dealers and Carriers, Street Vendors, and Counter Sales *(not mailed)*	0	0
(2) Paid or Requested Mail Subscriptions *(Include Advertisers' Proof Copies/Exchange Copies)*	1150	1066
c. Total Paid and/or Requested Circulation	1150	1066
d. Free Distribution by Mail *(Samples, Complimentary, and Other Free)*	10	10
e. Free Distribution Outside the Mail *(Carriers or Other Means)*	0	0
f. Total Free Distribution	10	10
g. Total Distribution	1160	1076
h. Copies Not Distributed		
(1) Office Use, Leftovers, Spoiled	140	224
(2) Return from News Agents	0	0
i. Total	140	224
Percent Paid and/or Requested Circulation	99%	99%

Signature and Title of Editor, Publisher, Business Manager, or Owner:
John D. Hall, President, October 1, 1994.

10th ANNUAL

SYMPOSIUM ON APPLIED COMPUTING (SAC '95)

• February 26–28, 1995 •
Opryland Hotel Nashville, Tennessee

"Business and Government Applications"

Sponsoring ACM
Special Interest Groups include:
SIGFORTH
Irving Montanez
Brookhaven National Laboratory
montanez@bnl.gov

Special tracks include:
Forth
Jack Woehr
jax@well.sf.ca.us

Symposium Chair
Jim Hightower
California State University-Fullerton
Mgmt. Science/Information Systems Dept.
Fullerton, CA 92634-9480
hightower@acm.org

Program Chair
Ed Deaton, Hope College
Department of Computer Science
Holland, MI 49422-9000
deaton@acm.org

Direct general inquiries to the conference chair or the program chair. Detailed information on special tracks, and mailing addresses of all track chairs and SIG representatives, are available from the program chair or by anonymous ftp from acm.org in the directory: ANONYMOUS.SIG_FORUMS.SIGAPP

For information about the sponsoring ACM SIGs, contact:
Hal Berghel
University of Arkansas
Computer Science Department
Fayetteville, AR 72701
hlb@acm.org

SAC is the annual conference of the ACM Special Interest Group on Applied Computing (SIGAPP). For the past nine years, SACs have become a primary forum for applied computing practitioners and researchers. SAC will bring together applied computer scientists from related ACM Special Interest Groups (SIGs): SIGAPP, SIGAda, SIGBIO, SIGCUE, SIGFORTH, and SIGICE.

SAC '95 will begin on Sunday, February 26, 1995, with pre-conference tutorials. Paper presentations, panels, and workshops will begin on Monday, February 27.

This announcement was edited for publication. For the complete call for papers and participants, and for more conference details, contact Irving Montanez or another of the individuals listed above.

Maryland FIG Chapter

Bliss Carkhuff's Radiation-Hardness Tests

The Maryland FIG Chapter had an interesting meeting recently. Bliss Carkhuff of the Johns Hopkins Applied Physics Lab brought, demonstrated, and explained his SC32-based system for observing integrated circuits in dynamic conditions under radiation bombardment.

Bliss has a delightful presentation style, and he brought a bunch of way neat stuff. The meeting didn't even begin to thin out for *four hours!* And at that point, it actually broke into microSIGs.

Forth has allowed Bliss to salvage expensive test time...

Bliss tests integrated circuits for radiation hardness. The use of Forth and the SC32 Forth-engine-based computer system he has devised have apparently been of great help to him.

Bliss' interest in Forth was sparked when he showed a large test board he was having a problem with to Marty Fraeman. Marty asked what the board was supposed to do. When told, Marty wrote a one-line Forth word with the same desired semantics as the troublesome test board. Bliss immediately "borrowed" an SC32 single-board computer Marty had, and obtained a copy of *Starting Forth*. Marty waited for a query from Bliss for some help with Forth. After several months without hearing from the new convert, Marty called Bliss and asked how the Forth was coming along. He was told that the SC32 SBC test rig would be used at the Brookhaven test site in a matter of days.

Bliss' system is based on an SC32 single-board computer, a backplane bus Bliss devised, several other half-eurocard boards for analog and the like, a frame for the DUT that mounts in the vacuum chamber (where the devices get bombarded), a board that mounts in the frame to hold the DUT, a suite of software tools from Silicon Composers, Bliss' extensions to those tools, cabling and power supplies, and a laptop clone.

Bliss' entire test rig fits in two flight cases. He can set it up in about 20 minutes. Other people who do similar testing at Brookhaven show up in station wagons, riding low. The multiboard computer goes into the test chamber with the DUT, greatly reducing what has to be run in and out of the vacuum chamber. Left outside the chamber are two or three small power supplies, the laptop, and a DVM. Dissipation of heat is a big problem in a vacuum, and Forth's efficiency seems to extend to the thermal variety. Forth has also allowed Bliss to salvage some expensive test time: when a DUT's behavior is wildly other than what was expected, Forth allows Bliss to cobble up new or modified tests while the meter is ticking, rather than wait for another trip.

The system is very Forth-like. It's nothing more than it needs to be, and lets Bliss concern himself with "draining the swamp." This aspect of Forth has affected the Maryland FIG Chapter meetings quite a bit. We often spend a lot of time on intriguing physics or math or whatnot, and have to consciously steer things back to Forth.

—Reported by Rick Hohensee
(hohenzay@tmn.com and rickh@cap.gwu.edu)

Julian V. Noble Crunches Cars & Numbers

Julian Noble talked to the Maryland FIG Chapter about scientific programming. He titled the talk "Forth isn't for number crunching, is it?" Here's a rough outline:

Julian is a physicist. He took up Forth when he was stuck using Fortran on a limited machine. "I could listen to my beard grow waiting for the compiler." He had prior Forth experience on the Jupiter Ace, and had a PC Forth with him that he was able to use to perform floating-point operations (even though it did not come with floating point). In essence, he used Forth as a way to control the 8087 coprocessor through a calculation.

Scientists and engineers need floating-point arithmetic, both real and complex. Scaled arithmetic á la classic Forth is not good enough because, for many problems, the dynamic range can be 30 orders of magnitude. Floating point is at least as fast in hardware as scaled integer arithmetic on the CPU, so there is no reason not to use it. It is cheaper to buy a chip than to develop software floating-point code.

He bought a Forth package from Harvard Softworks; it ran very fast. It was easy for him to add complex numbers and sophisticated matrices, things which were not so simple before. And, of course, he got more control of the machine.

As he got better at Forth, he found out how to optimize for speed. He already knew from Fortran to take things out of inner loops and factor out repeated expressions; now he started coding routines in assembly. He did it for the sense of power, because he *could* do it. He over-used the assembler because it was so easy to do. Now, sometimes he wants to port those early routines elsewhere and has to reverse-engineer the assembly code.

Julian noticed that a lot of his time was spent evaluating complicated algebraic expressions. He translated each expression from infix notation into postfix by hand, and it was easy to make mistakes. The factoring that aided the programming didn't match up with the sort of factoring that helped him understand the problem, so that factoring, per se had limited value. More valuable tools were special data structures and functional notation. Too much detail in the form of "noise" words—@, !, >R, R>, etc.—obscures the main pattern, so he learned to hide some details. (He pointed out that advances in physics often have come from improved notation that unclutters central ideas, e.g., Maxwell's Equations.) He came up with array notation that worked for him:

```
1array Y{ 2array M{{
Y{ I } M{{ I J }}
```

Still, the debugging was tedious. Julian noticed that he included the original infix expression regularly as a comment. How much better if that comment could be his code! He wrote a parser he called a FORmula TRANSlator. It was designed simply, to do the job he needed without lots of extras to go wrong.

It's built around a software floating-point stack. Each item takes 18 bytes; that can hold the largest item he needs (double-precision complex) with two bytes left over to hold the data type. Some audience members objected to the wasted memory: some of his items, after all, would fit into four bytes. Julian explained that he rarely needed more than 40 items on the stack, so the space used was insignificant. He saved a lot of his execution time not having to worry about stack widths. (He first did a variable-width stack, and the speedup was a factor of 5–10.)

Keeping the data types with the calculations, his code could do mixed-mode arithmetic without needing his attention at all. Variables give warning messages when they don't have room for the items being stored. Two bytes is a lot of space for data types when he only has four scalar data types, but this is also unimportant.

Some audience members pointed out the inefficiency of constantly removing items from the hardware stack in the coprocessor to the software floating stack, only to move them back.

Julian said, "Some systems have an unlimited depth floating-point stack that extends the coprocessor's eight-deep stack into memory. I wrote one of those about six years ago. But the coprocessor chips do not have any automatic push/pop or other intrinsic 'hooks' to extend their stack into RAM, hence there is a lot of bookkeeping to do it this way. Eventually I came to realize that it was better to keep the floating-point stack in RAM, with just the TOS on the math chip (this works well for the Motorola and Weitek series of chips also). This is analogous to some Forths that keep the top of the data stack in the BX register, as a one-deep cache. Studies show one can eliminate all but a few pushes and pops this way."

He was asked what optimization his parser did. It didn't do much optimization. The most important ways to improve code are to remove things from inner loops and to factor repeated expressions, and these are things that programmers learn with their mother's milk. Then optimizing compilers grind away looking for the inner loop items that aren't there.

It seems that, in scientific computing, it's often more important to get quick results than to get quick computation. A program might be used one to ten times before being discarded or modified; it's much more important to get it to run right than to get it to run fast. Forth's primary competitor in this arena is not C but Fortran, and its most formidable competitor may be Visual Basic—easy to use and quick results, though without Forth's flexibility. Many people try to use Mathematica or Maple, and find them very slow.

Julian figured that if he did need to speed up his code, he'd do better to write an inner loop in assembler than complicate his parser. He gave an example from solving large systems of linear equations. The inner loop (of three) multiplies a row by a constant multiplier and subtracts it from another row. This is just a few instructions' worth of assembler, hence easy to write and debug. A program with just this loop optimized is hardly longer than the unoptimized Forth, yet for large problems runs at the intrinsic speed of the silicon.

He presented an example, a set of six simultaneous first-order differential equations. It modeled an automobile sliding sideways on pavement toward a curb. When the car hit the curb, it would flip over the curb and crunch beyond it (if the collision was inelastic) or bounce up and rotate about its center-of-mass (if the collision was elastic). The parameters were adjustable, to simulate sliding on dry pavement or ice, or to make the collision between tire and curb elastic or inelastic.

The graphics were not fancy, but they showed very well what was happening. When he showed us the code, there were about a dozen pages of parameters with comments, and toward the bottom were his six equations on one page. His parser had almost eliminated the complicated coding and debugging process! It's clear that the main remaining limiting factor is handling those parameters.

His parser had almost eliminated the complicated coding and debugging process!

Julian had special code that tested energy buildup; when the energy of his system increased rather than dissipated, he could figure he had a term in the equation with the wrong sign. (Obviously, it's more important to get the signs right than to optimize for speed.) This kind of debugging was very simple using his flexible interactive system written in Forth.

Julian noted that developing and commenting the Forth code for the car rollover simulation took about two days. His experience with Fortran leads him to believe the time would have been at least a week, even with a fast workstation and a fancy debugging environment. Graphics would have taken longer, which is why accident-reconstruction experts often use Fortran codes to produce numerical results which are then fed to drawing programs such as AutoCad for animation. The expense, complexity, and opportunity for something to be wrong in such jury-rigged systems is beyond imagining.

Julian Noble is doing several things intended to help spread the use of Forth. His book *Scientific Forth* is still available, and people bought copies at the meeting. He's deeply involved in the thrust to build ANS Forth scientific libraries. He's writing a Forth textbook with Brad Rodriguez, intended primarily for E.E. departments. He doesn't know which things will work to make Forth more popular, and figures if we do enough different things we'll find out. "When you shoot a shotgun you don't know which pellet will bring down the bird."

—Reported by Jet Thomas
jethomas@genie.geis.com

("Letters," continued from page 6.)

original. The elimination of the *; ;* also prevents the error of putting code between the final *; ;* and the `ENDSWITCH`.

Finally, I've managed to implement the whole thing with, in terms of the '79 and '83 standards, only one non-standard word, `SWAP/C`, which swaps elements on the control stack. No doubt `SWAP/C` is just an alias for `2SWAP` in most Forths and, in fact, Rottenkolber uses `2SWAP` directly, but this assumption cannot be depended on. In one Forth I know, with no so-called "compiler security," `SWAP/C` would be simply `SWAP`; in Forths with a separate control stack, neither of these solutions would work. `?<MARK` and `?>RESOLVE`, while possibly common, are not standard and are not in any Forth I use. Presumably, my switch would be ANSI standard (except for that one implementation word) with the substitution of `POSTPONE` for every `COMPILE` and `[COMPILE]`, but I don't have an ANSI Forth to test it with.

Anyway, whether SWITCH is a valuable or frivolous contribution to Forth, I am indebted to Rottenkolber for showing it could be done, and for inspiring me to try to do it better. *[See Figure Two, next page.]*

Sincerely,
Richard Astle
P.O. Box 8023
La Jolla, California 92038

Making FIRE

Hello Forthers.

I have been informally proposing a project for whomever is interested in developing a coherent Forth-based consumer or hobbyist platform. I have been posting periodic iterations of a general outline of the design, as I see it, in comp.lang.forth (the Internet USENET newsgroup), and Jet Thomas handed out some copies of it at Rochester 94. The project has acquired the name FIRE, as in:
: FIRE the Individual's Recursively Forth Environment ;

Interest in FIRE has been limited but persistent. The crucial idea driving FIRE is that Forth clashes with how software is usually sold, and thus Forth should be sold or distributed as a unified system of hardware, docs, and software, perhaps with a business mechanism to compensate Forth authors on a word-by-word basis. FIRE also proposes a modular enclosure and power supply, an ANS Forth-based single-user, preemptive multitasking OS, a three-stack Forth dialect called Bana as an optional vocabulary, and lots of other stuff. If you find this interesting, please contact me. FIRE also now has a mailserver. E-mail fire-l@artopro.mlnet.com with a subject line of SUBSCRIBE to participate in the FIRE discussion via e-mail.

Rick Hohensee
P.O. Box 11340
Washington, D.C. 20008
rickh@cap.gwu.edu

Figure Two. Astle's cleaner switch.

```
\ A CLEANER SWITCH             RA 21SEP94
\LMI WinForth implementation but completely 83-standard
\ except for implementation-dependent SWAP/C
EXISTS? cases? .IF FORGET cases? .THEN
\ compilation flags
VARIABLE cases?
VARIABLE default?

\ control-stack operator
\ with compiler security two items are put on stack for each
\ control item
: SWAP/C 2SWAP ;  \ with compiler security
\ : SWAP/C SWAP ; \ without compiler security

\ these non-standard operators should be coded directly
\ in terms of the non-standard factors MARK and RESOLVE
: LEAP  COMPILE FALSE [COMPILE] IF ; IMMEDIATE
\ : AGAIN       COMPILE FALSE [COMPILE] UNTIL ; IMMEDIATE

\ switch like C
\ compile-time stack diagrams:      cases OFF | cases ON
\ except for ;SWITCH which shows: default OFF | default ON cases
\ if and leap addresses are treated the same thus begin/leap
: SWITCH:
        cases? OFF default? OFF
        [COMPILE] : ;
: CASE(         ( | if/leap --- | else )
        cases? @ IF [COMPILE] ELSE THEN
        COMPILE DUP ; IMMDIATE
: )IS           ( | else/leap --- if )
        COMPILE = [COMPILE] IF COMPILE DROP
        cases? @ IF SWAP/C [COMPILE] THEN THEN
        cases? ON ; IMMEDIATE
: DEFAULT       ( if | --- begin if | begin leap )
        cases? @ 0= IF [COMPILE] LEAP THEN
        [COMPILE BEGIN
        SWAP/C
        default? ON cases? ON ; IMMEDIATE
: ;SWITCH       ( if | begin if/leap --- )
        cases? @ 0= IF CR ." NO CASES COMPILED " KEY DROP QUIT THEN
        [COMPILE] ELSE COMPILE DROP
        default? @ IF SWAP/C [COMPILE] AGAIN THEN
        [COMPILE] THEN
        [COMPILE] ; ; IMMEDIATE
: BREAK COMPILE EXIT ; IMMEDIATE

\ EXAMPLE
SWITCH: DOG
        CASE( 1 )IS CR ." ONE    "
        DEFAULT     CR ." DEFAULT"
        CASE( 2 )IS CR ." TWO    "
        CASE( 3 )IS CR ." THREE  " BREAK
        CASE( 4 )IS CR ." FOUR   "
        CASE( 5 )IS CR ." FIVE   "
;SWITCH

CR .( SWITCH COMPILED )
```

Guest Essay

Object Code vs. Metacode

Andreas Goppold
Neubiberg, Germany

A couple of short definitions:

"Object code compilation" is the software paradigm under which the formal instructions for solving a problem (a computer program) are translated—using appropriate tools like compilers—into the object code (or native binary code) of a computer on which, then, the solution of the task is to be worked out.

"Metacode programming" means that a (native object code) control program is (permanently) resident in the working memory of the computer, and this program processes the formal instructions by direct interpretation.

The Forth community has a great potential in the know-how of metacode application.

Metacode programs are a mix of the Von Neumann categories of program and data. This dual nature contradicts the clean division that has developed for the purpose of scientifically and methodically handling the task of programming. The fact that metacode is instructions as well as data, however, allows a completely different approach to the subject of programming. In the computer world today, we experience a clash of paradigms between CISC and RISC.

Analogous to the RISC/CISC polarization, the history of the computer industry has seen quite an interesting battle between the advocates of object code and metacode. This battle was interesting, alright, but quite one-sided. Since up until now only the processing speed of the computer, i.e., the optimum utilization of the machine, stood in the foreground, the winners have always been the proponents of object code. Thus, a few systems have been left behind, others have been interesting academically and fruitless commercially, and a few have enjoyed quite a busy life in quite unknown niches: UCSD-Pascal with the P-Code machine, Smalltalk with the Bytecode machine, the PICK system, Forth and a few variants like Actor and Amber. LISP can be added to the count, as well as APL and Mumps; likewise, many BASIC systems that compile incrementally to tokenized code. Today, the best known and most widely used metacode system is Postscript.

Why have there, time and again, been such defenders of a paradigm which did not have a chance in the view of the processor economy? It is easy to see, when one knows a few such languages, why programmers who have had experience with such a system are reluctant to let it go (see also *Die Gaensekueken* ["the baby ducklings"] by Konrad Lorenz). Metacode systems decisively lighten the lives of programmers. They shift a major part of the complexity of the programmers' work to where it belongs—the computer. Amazingly, the beginnings are as old as computer science itself, for instance APL or LISP. The good ideas are very old, but today the time has finally come when they may celebrate their resurrection. The reason is, again, to find balance in the present shift of the technology.

Within approximately the last ten years, the balance has decisively shifted to where the limiting factor is no longer the machine, it is now the human factor. Certainly, the larger part of the computer industry has been awakened to this knowledge with the catchwords "software crisis" and "downsizing." One might say that the Macintosh computer was the torch to this development. When the first version of this machine was made, 90% of the processor's efforts was concentrated on the user interface. (At least with the first "toaster." The 68030 models do leave a little spare power.) There was no programming interface; for that, one had to use the Lisa.

Twenty years ago, that was unthinkable, and has become possible today because something like it is available at prices starting at $2000, since Apple has had to follow the price plunges in the PC area. A further representative of this philosophy is NeXT (from Steve Jobs, the father of the Macintosh, of course). What is good for the user is good for the programmer. NeXT is definitely on the way to giving the programmers the same quantum leap as the Mac has for the lay user. And NeXT uses for its display paradigm the Postscript system (metacode), in contrast to the industry's quasi-standard, X-Windows.

The Opportunity

I see something like a historic opportunity. The computer industry is in a deep crisis (the rest of the world apparently not less so). Solutions are urgently needed and are paid for dearly. The Forth community has a great potential in the know-how of metacode application. It would be possible to organize a comeback of the Forth idea if the Forth community were to turn to the present problems of the computer industry in a coordinated and constructive manner.

To that end, though, a few prerequisites are needed. In order to conquer new territory, it is necessary to unload some ballast. And such a decision can have difficult consequences. Not everyone wants to make the move. In order to find entry into the computer world, the chains to the specific appearance of Forth would have to be broken. The important things here are the structures that lie under the surface of Forth. The token-list interpreter. The possibility, for instance, via token threading to set up code of minimal size. Postscript is proof that token-coded systems have a future. Postscript itself is everything but efficient or elegant, but it is the best solution to a certain class of problems.

If *Vierte Dimension* really wishes to be interesting to readers outside the Forth community, the narrow adherence to the conventional image of Forth must be given up. The subject "metacode systems" is hot and is in the market with a future. A magazine that concentrates on this subject will find a wide audience. When the Forthers learn that they are experts of a certain kind of metacode systems, they will suddenly realize that their knowledge is very desirable.

I am convinced that my past analyses have always been correct and that I have the ability to recognize things from the historical perspective, while they still are behind the horizon of consensus reality. Forth wasted its historical chance fifteen years ago. According to all natural laws, Forth would follow the road of all extinct branches of evolution. Here is another possibility. It cannot proceed under the name of Forth, but the substance can remain.

Andreas Goppold is a contributor to *Vierte Dimension*, where the original, German version of this essay was published.

dot-quote

"[Forth] supports a modular,
bottom-up style of programming that
I personally find to be highly productive.
It is easy to port, it is self-contained,
it will run on minimal hardware
configurations.
It is *not* a philosophy of life
or a cure for cancer
or a way to transcend mundane reality
and merge with the Tao.
Appeals to mysticism do not help
the credibility of Forth."

—Ray Duncan on comp.lang.forth
Used with permission

ADVERTISERS INDEX

(Fast Forthward, continued from page 39.)

which handles queries of the dictionary namespace, or symbol table. By accepting a string as input and returning an execution token, `FIND` lets other routines ignore the dictionary structure. Unfortunately, `FIND` will need even more flexibility to support modules.

The way `FIND` works is to test for a name match as part of a loop that handles one dictionary entry with each iteration. Obtaining flexible behavior from a loop-containing routine can be problematic. The loop is a form of barrier or obstacle, like a two-lane merge at a freeway on- and off-ramp. Leaving the loop early often requires special (loop-finalization) processing, such as stack-value shuffling. Name visibility is directly related to the continuation, normal termination, and early termination of the search loop.

If we want industrial-strength modules, we must dip into the murky details of loop termination and loop continuation conditions. Nevertheless, it is better to refine one version of `FIND` rather than create multiple versions to be used in tandem. (The fewer "delicate" loops we create, the better off our systems and applications will be.)

Shortly, I will describe the approach I took to impart more flexible namespace management to Forth.

By the way, vocabularies are a neat trick that helps make namespace management more flexible without any impact to the `FIND` routine. A little bit of vocabulary state goes a long way toward modifying Forth's search behavior, yet the search algorithm in `FIND` remains unchanged. The use of a richer set of data structures in lieu of a more complex algorithm is a programming technique worth remembering. (In a similar vein, the next installment suggests several new data attributes for each dictionary entry.)

Even when modules aren't the objective, there are plenty of uses for a more flexible FIND.

Streamlining Through Vector Parameters

Setting aside the `FIND` routine for now, consider how a generic sort routine uses well-chosen routine parameters to permit the processing performed at each loop iteration to be determined by its calling context.

A generic sort routine is passed a couple of parameters that refer to routines that can be characterized as "sorting extensions." Only two sorting extensions are required for each run of the sort routine. Those extensions may be selected differently at each new calling context.

One parameter for the generic sort is a pointer to a comparison routine applicable to the type of data values to be sorted. The other parameter is a pointer to an element-exchange routine applicable to the array to be sorted. Such parameters allow the sort problem to be factored into a number of sorting extensions.

By leaving any unneeded extensions out of a program, program compactness is achieved. Nevertheless, if new types of sorts are eventually needed, there is no major design task to perform. The original design is flexible. You are able to reuse code and, thereby, curb program growth. Essentially, a *framework* is formed by the generic sort that accommodates future expansion very efficiently.

To make such a technique work, we must be able to specify the correct extensions for the routine at compile time (which avoids control-flow logic to support run-time decisions about which action to take).

We may be tempted to create new instances of iterative routines rather than attempt to craft a single routine to satisfy our various needs. Yielding to this temptation leads to a number of similar-but-related routines. It also creates fatter applications.

The generic sort is a good example of serving different usage contexts with suitable functionality while using a bare minimum of code resources.

Streamlining Through Iteration-Control Parameters

The use of execution vectors as interface parameters lets the calling context for a generic routine select suitable processing. For a way to manage the iteration process based on the contextual system state, a slightly different approach is required. Parameters that help control loop continuation and termination conditions can be passed, as will be described in terms of a new version of Forth's `FIND` routine.

For module support within `FIND`, the contextual state can help determine whether a routine is currently visible. If we are trying to access a private routine of a module from a context outside of the module, we should not be able to find it.

Such access denial need not stop `FIND`'s iterative search processing. In such cases, `FIND` can continue to search for a routine of the same name in a place other than the private portion of a module.

Suppose the calling context is changed. Suppose a reference to a routine is specified in the same module as the routine being defined. In that case, `FIND` should cease iterating upon the first name match in the module and exit after leaving the associated execution token on the stack.

Management of iteration behavior based on a usage context state can be achieved through passed parameters. A version of `FIND` can be defined to test new word-specific (state) information against a passed parameter.

For module purposes, `FIND` can accept as a new input the module location where a new routine is being defined. The location information for the dictionary entry currently being examined can be compared to the passed parameter.

Even when modules are not the immediate objective, there will be plenty of uses for a `FIND` that takes two new input parameters for added flexibility. As an example of other attributes that could be taken into account, consider immediate words. Should immediate words be located in non-compiling states? Locating and executing these words at such a time permits needless errors, often without any error indication. Not finding those words will help correct the situation.

You could say that a better kind of error could be permitted ("xxx not found"). That way, error recovery is equivalent to error detection. Otherwise, inappropriate compiling actions may have to be undone.

Such a FIND routine can exit after returning the token for a matched string, but only when it locates a Forth word with a particular attribute determined by context. Otherwise, it must continue searching for another word with the same name and the desired attributes.

The new FIND can also continue searching (iterating) if it locates a Forth word with the correct name but with an undesired attribute for that search context.

The first scenario involves exiting FIND successfully due to the presence of a context-defined *word-admission* attribute. The second scenario involves continued iteration (searching) based on the presence of a context-defined *word-rejection* attribute, possibly leading to a "no-match" exit status. Put another way, search-loop iteration would continue (and the currently matched word would be rejected) if the admission attribute was missing or if the rejection attribute was present.

The new FIND takes a *word-admission* parameter as well as a *word-rejection* parameter. By passing zero as the *word-rejection* parameter, the routine concerns itself with the presence of a *word-admission* parameter only. By passing zero as the *word-admission* parameter, the routine pays attention to the *word-rejection* parameter only. With that design, the calling context could establish the exact loop-termination conditions that ought to prevail.

As sought, a single routine should take the place of separate routines that would otherwise be needed to support new search criteria.

Function Wrapper Interfaces

An obvious flaw in this approach is that the new FIND was no longer standards-compliant. My remedy was to make another FIND routine that served as a wrapper function calling my "more primitive" FIND, which I renamed ?FIND.

The FIND wrapper requires the inputs of a conventional FIND routine. It then supplies each of the additional, nonstandard parameters required by ?FIND. Outwardly, standards compliance is met. Inwardly, it is a circuitous fulfillment of the standard—but one that suited my own purposes well.

Through a careful selection of control parameters, my solution pared down the number of routines containing a similar search loop. Through function wrappers, it also provided an efficient way to include some nonstandard features in an otherwise standard system.

By factoring the standards-compliance code into a separate wrapper routine, I preserved an optimal solution for my needs. The wrapper function supported the expected programmer interface with a tad more overhead.

The function-wrapper solution does not compromise the high productivity of experienced Forth programmers. It also allows novices to trust generic documentation to describe the Forth system, at least in terms of the basic (standard) Forth features that it supports. Meanwhile, both types of users can derive benefits from the nonstandard functionality that was unobtrusively and efficiently included. (The next installment will clarify this point.)

The benefits of the function-wrapper approach far outweigh the slight performance penalty in routine-calling overhead. As mentioned earlier, two different styles of interface are able to efficiently coexist.

Forthward Ahoy

The use of function wrappers helps us move beyond standards while remaining backward-compliant with them.

So an ANS Forth standard does not have to be the endpoint of the evolution of Forth. A newly accepted standard merely offers a new point of departure. You can use the standard as a jumping-off point to help take you where you want to go—or you can create a better starting point and make the standard one of scenic stops that your system visits along the way to your true destination.

Function wrappers are useful for interfacing standards-compliance routines to application-specific routines. The nonstandard routines are freed from standardization constraints—so they can incorporate added functionality to suit special needs.

The words of another *Forth Dimensions* author are appropriate to leave you with. Strive for "sophisticated simplicity."

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

A Reconciliation with ANS Forth and an Exercise in Interface Design

Mike Elola
San Jose, California

In the May/June installment of this column (*FD* XVI/1, "Rapid Development Demands Quality Interfaces"), I began what was supposed to be a multi-part discussion about the design of interfaces. I allowed that discussion to be derailed by an intervening discussion about modules and modularization tools.

In the original discussion, I claimed that Forth's reputation as a productivity tool depends on the development of well-interfaced code. I was responding to Byron Nilsen's article about the sometimes-difficult F83 vocabulary mechanism.

The intervening discussion thread was inspired by Leo Brodie's comments in *Thinking Forth*. While Brodie touted components arrived at stylistically, I argued for the use of formal modularization tools.

These distinct topics of discussion will eventually converge: the added rigor of modules will require building a canonical interface between modules. The module interface must offer one protocol for binding routines that cross module boundaries and another protocol for binding routines within the same module. The availability of module routines must vary according to their designation as part of a module's interface. That way, the interface routines for a module can properly encapsulate (hide) any private code and data.

If we want industrial-strength modules, we must dip into the murky details...

For module support, Forth's dictionary search protocol must be expanded. Before implementing real modules, certain evolutionary steps can be taken that stop short of that goal. These intermediate steps involve simple but valuable refinements to Forth. (Stay tuned to the next installment for the majority of the details.)

An exploration of these shorter-term goals will be carried out in this and the next installment of Fast Forthward. At the same time, this installment continues the discussion of interface design from the point where I left off several issues back.

Benefits of Routine Flexibility

We should not overlook how a few flexible routines can accomplish the work of several, less-flexible routines. A few well-crafted routines can offer more robust operation, greater reuse, and more code compactness. Benefits like this already make Forth a high-productivity programming tool.

Code flexibility and streamlining go hand-in-hand. I think of them as the ability to obtain many different useful behaviors from a bare minimum of code resources. Such goals are often elusive. Therefore, we should try to codify design guidelines to help us achieve those goals.

The code-streamlining techniques I will be describing suggest ways to impart flexibility to routines that incorporate loops. The applications we create should be as streamlined as possible. The danger of ignoring such design issues is that an application can easily balloon in size and complexity.

Curbed program growth is the chief benefit of flexible, easily reused code. While programmer productivity is the official chant of the promoters of code reuse (the champions of OOLs), the apparent increase in productivity is probably due to less code being written. (Of course, there are those who refute the productivity claims made for object-oriented languages.)

In any case, less program code reduces development costs throughout the lifecycle of an application due to decreased program complexity and increased program maintainability.

Flexibility at the Point of Interface

When it comes to the interface, flexibility is practically a requirement. The flexibility of an interface profoundly affects routine reusability. Inflexible interfaces may cause a proliferation of basically redundant routines. For this reason, the interface should be crafted with care.

To gauge all of the possible usage contexts for a routine, an overview of all the routines to be included in an application is needed. Accordingly, a distinct design phase is required before coding shifts into high gear. (I am incredulous when I hear it said that Forth programmers can skip directly to the coding step without first developing a design.)

More Corrections to the ANSI Standard Forth Quick Reference

Ask and ye shall receive. I asked for corrections and received more responses than any other request for responses to date. It pleases me to know that the card is receiving some attention. I also hope it helps reduce the page-turning you do to refresh your memory regarding various ANSI Forth details.

Dr. Wonyong Koh and Marty McGowan both wrote to say that `FORGET` belongs with the TOOLKIT EXT wordset rather than the CORE wordset. Note also that TOOLKIT is used in the quick reference card, which is equivalent to TOOLS in the standard.

Zsoter Andras was confused by a mistake regarding `WORDLIST`. The quick reference card falsely indicates that `WORDLIST` consumes a word from the input stream. The `WORDLIST` routine does not have any input requirements, neither from the input stream nor from the stack. Strike out the line describing the effect of ***name*** immediately after `WORDLIST`. Change the description of `WORDLIST` to "Create a new wordlist, push its ID." Also, change the stack diagram to show that the execution of `WORDLIST` leaves a wordlist ID on the stack.

(To associate a name with the returned ID, we'll have to execute a real word-defining routine such as `CONSTANT`. You could define your own routine that is both a word- and wordlist-defining routine, but please don't name it `WORDLIST` for faithfulness to the reference card's error!)

Dr. Wonyong Koh also noted my misnaming of `EKEY?`, `EMIT?`, and `KEY?` on the quick reference card as `?EKEY`, `?EMIT`, and `?KEY`. Please note these changes on your cards.

Product Watch

JUNE 1994

Offete Enterprises and Computer Cowboys announced the initial availability of the high performance MuP21 CMOS multiprocessor chip. It features coprocessors for memory and video that make it suitable for use in computer displays, CAD systems, communication systems, video games, and embedded control applications.

The MuP21 executes a set of 24 native instructions, each requiring five bits for storage. The memory coprocessor fetches 20-bit memory words from dynamic RAM at 20 MHz. Because each memory word can contain up to four native instructions, the memory subsystem is able to provide instructions to the CPU at a peak sustained rate of 80 MIPS. This manner of addressing the performance bottleneck arising from slow DRAM (relative to microprocessor speed) is ideally suited to keeping chip-manufacturing costs low—and keeping chip power demands low.

The independent, on-chip video coprocessor generates an NTSC signal based upon 16-color images held in DRAM.

A minimal target system might use an MuP21, five 1Mx4 DRAM chips, and an eight-bit boot ROM. To use the video coprocessor, an external crystal clock must be added for NTSC signal synchronization. Otherwise, no external clock is needed.

The MuP21 Development System includes a single-board computer containing an MuP21 chip, 2.5 Mbytes of fast DRAM, a 128K battery-backed SRAM PCMCIA memory card for booting and mass storage, and an 8255 parallel-interface chip for I/O. It also includes a PC-based target compiler and a simulator that allows a PC to run compiled MuP21 instructions in an emulation mode.

COMPANIES MENTIONED

Offete Enterprises, Inc.
1306 South B Street
San Mateo, CA 94402
Fax: 415-571-5004
Phone: 415-571-8250

With respect to an interface, a change is synonymous with an interface failure. Like a dam that breaks when a threshold of pressure has been exceeded, a broken interface leads to a flurry of changes.

An interface may function perfectly well in many different contexts. Nevertheless, it will be rendered defective if it's unable to handle some new need. This is what happens to `FIND` when module requirements are introduced. Modules will spur the development of a new version of `FIND` that can handle new dictionary-search criteria that suit module needs.

A flexible interface can be considered a form of change insulator. However, with the introduction of a new requirement that the interface can't handle, its change-insulation role breaks down. If the interface is allowed to change, corresponding changes usually must be conveyed throughout the system or application.

The alternative of using a new routine creates redundancy and inefficiency, although it can perhaps circumvent changes to some of the existing code. The best design almost always seems to take more effort. It might involve discovery of distinctly different usage contexts before coding begins (in the design phase). It might involve a willingness to make mass changes to suit interface alterations for a routine that is already in place.

In the case of `FIND`, kernel changes can be expected that will force regeneration of the Forth kernel. However, if a Forth system is being built from scratch and you anticipate the need for module support, no such "breakage" need occur in the first place.

(The use of a function wrapper will also be discussed. It can allow several interfaces to coexist for the same services, without redundant code.)

Namespace Management Flexibility

Forth's namespace visibility is determined by `FIND`,

(Continues on page 36.)

ATTEND

for the sixteenth annual and the 1994

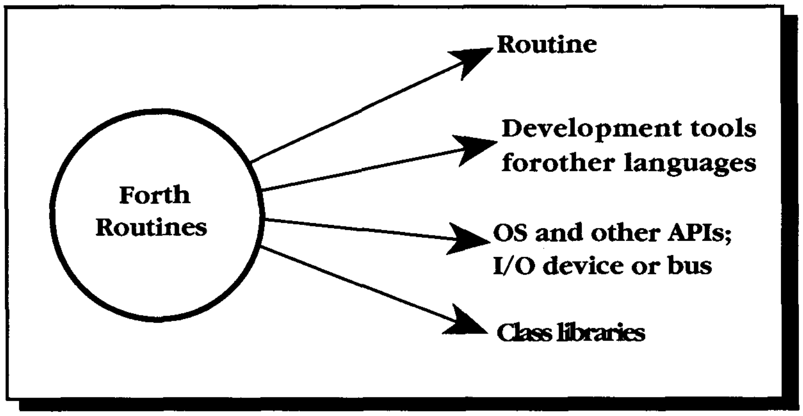
FORML CONFERENCE

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

"Interface Building"

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

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


Forth Routines
Routine
Development tools forother languages
OS and other APIs; I/O device or bus
Class libraries

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

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

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

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

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

Advanced Registration Required

Call Forth Interest Group Today, 510-893-6784

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

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

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

Mike Elola, Conference Chairman — Robert Reiling, Conference Director

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