$10 Volume XIV, Number 5 January 1993 February

FORTH DIMENSIONS

Forth-based Message Service

Graphics & Foating Point

Optimization Considerations

Principles of Metacompilation (III)

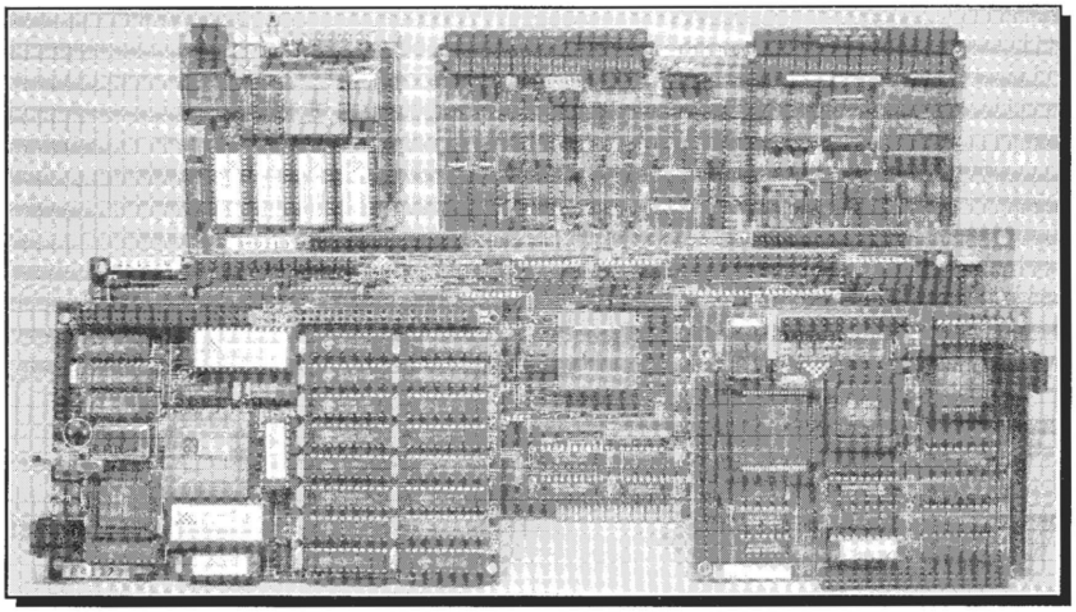
Embedded Systems Conference

SILICON COMPOSERS INC

FAST Forth Native-Language Embedded Computers

DUP

>R



C@

R>

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

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

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

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

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

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

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

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

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

Contents

Editorial

Forth's continued recognition—for instance, as a tool of choice for engineers, for developers of embedded systems, and for scientific instrumentation—is fine, as far as it goes. But successful efforts on a larger scale may be less appreciated, perhaps not only because they are fewer, but because they are less discussed or do not fit the Forth stereotypes. Such applications may be characterized by large amounts of code, multiple programmers, complex interface requirements, volume of data, etc. The one that first comes to mind for many people is the international airport implemented in Forth in the Middle East; it was talked about at the conferences a few years back, impressing us all with the scope and prestige of *that* contract. The next one most people think of is—

I'll bet most people draw a blank. Maybe the designers, engineers, programmers, and managers who work on these things are so absorbed in the next challenge that they aren't spending a lot of time bragging. Maybe the reports are true that some companies think of Forth as their secret advantage, which would explain their silence on the subject. If I hadn't dined at a certain table at a particular conference, I would never have heard of the telephone system which, after a few months' protoyping in Forth, had already outstripped a years-long effort using other languages.

As a gesture intended to bring attention to Forth's reliability, maintainability, and efficiency in large and/or complex systems, *Forth Dimensions* and the Forth Interest Group sponsored a call for articles about "Forth on a Grand Scale." Printed in this issue are two entries the referees believed best addressed the theme, albeit in different ways. Olaf Meding's first-place paper discusses Forth's contribution to the commercially successful EVE messaging system. Mark Smiley writes about mathematics and fractals in Forth. We thank them , and are pleased to present their work here. Articles about Forth in big places are always welcome!

* * *

With that concluded, *Forth Dimensions* is able to announce its next contest for Forth authors. In keeping with the theme of the upcoming FORML conference, we are seeking articles about "Forth Development Environments." The three articles judged best will be paid cash prizes, with the first-place author receiving $500. Some specific subjects in keeping with that theme (e.g., libraries, user interfaces, source and run-time browsers) are given in FORML's announcement on the back cover of this issue, although authors should not feel restricted to that list as long as they address the general theme. Deadline for submissions to this contest is August 1, 1993. (See the ad on page 25.)

* * *

Concluding in this issue is Brad Rodriguez' series of articles on metacompilation. His structured presentation of the topic is the most complete and comprehensible I can remember. Still, this is the kind of knowledge that only gets burned into our long-term memory when we apply it. Besides which, as the author would point out, specific choices and techniques may be affected by the particular application and environment. So re-read Brad's three-part series and prepare to take the next step...

...which begins with this issue's "On the Back Burner." Columnist Russell L. Harris presents a schematic and parts list for a relatively simple device readers can assemble themselves and use to explore the principles of metacompilation and embedded programming, with Harris' continuing guidance in future columns.

(P.S. Short of taking an E.E. course or bluffing your way in over your head at a new job, this is an excellent opportunity to get hands-on-hardware experience—make the most of it! But if you are more interested in programming such devices than in building them, call the Forth Interest Group to inquire about the availability and prices of partially or completely assembled boards; parts kits might also be offered.)

* * *

Gary Smith served long and well as *FD*'s "Best of GEnie" columnist, in addition to his Forth RoundTable duties on that database/communications service. He volunteered much energy and time tracing the threads of interesting and important on-line discussions for our benefit. Gary recently retired from this activity, and we wish him the best. Thanks, Gary, for your contributions!

—*Marlin Ouverson*
Editor

Forth Dimensions
Volume XIV, Number 5
September 1992 October

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

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

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

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, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621."

Letters

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!

Wanted: Visible Forth

I read the letter from Mark Martino ("Visible Words & Ugly Complexity," *FD* XIV/4) with interest. As always, Mark's ideas are creative and useful. I hope he pursues the idea of creating a "visible" Forth development environment. I would like to buy such a beastie from him.

I think Mark's approach would improve Forth documentation efforts, as well. Using "word boxes" might solve some of the issues addressed by Mike Elola in the same issue. (By the way, thanks, Mike.) Possibly, the visual environment can include Forth help files slaved to the visual word boxes. This would include parameters required by and output by the word, examples, etc.

Thanks for your attention.

Gus Calabrese, President
WFT
4555 East 16th Avenue
Denver, Colorado 80220

One Simple Syntax, Please

Dear Mr. Ouverson,

After reading Mike Elola's article ("Styling Forth to Preserve the Expressiveness of C," *FD* XIV/4), I just had to disagree. This is yet another attempt to improve Forth's image by turning it into the popular language of the day. We've been through this before. Forth as BASIC. Forth as Prolog. Forth as Lisp. Forth as God knows what.

I don't consider C's three capricious syntaxes an improvement over Forth's single, simple one. It's richer in the same way that government gobbledegook is richer than a Hemingway novel.

In an article recently published in *The Computer Journal* (#57, Sept./Oct. 1992), I explored the Shellsort in depth. As part of the sort engine, I wrote the following code:

```
: SHELL  ( -- )
  SETGAP  BEGIN  DECGAP
  ITEMS @  GAP @  DO
    I DUP S@  SV !
    BEGIN
      DUP GAP @ - DUP 0< NOT SWAP
      S@ SV @ > AND
    WHILE
      DUP GAP @ - TUCK S@ SWAP S!
    REPEAT
  SV @ SWAP S! LOOP
  GAP @ 2 < UNTIL  ;
```

I was rather proud to have squeezed it into a single screen. Today, I would have written the same code as:

```
: SHLMATCH? ( i -- i f ) \ f= true = no match
  DUP GAP @ - DUP 0< NOT \ Within array?
  SWAP S@ SV @ > AND ; \ Compare 2 array values

: SHLGETNEXT  ( i -- i' )
\ Shift value and get next index
  DUP GAP @ - TUCK S@ SWAP S! ;

: SHLCOMPARE  ( i -- i' )\ i = array index
  BEGIN SHLMATCH? WHILE SHLGETNEXT REPEAT ;

: PICKUPITEM ( i -- )
  S@ SV ! ;

: INSERTITEM ( i -- )
  SV @ SWAP S! ;

: SHLSHUTTLE ( -- )
  ITEMS @  GAP @  DO
    I DUP PICKUPITEM  SHLCOMPARE
  INSERTITEM  LOOP ;

: ?ENDGAP ( -- f )
  GAP @ 2 < ;

: SHELLSORT ( -- )
  SETGAP BEGIN
    DECGAP SHLSHUTTLE ?ENDGAP UNTIL ;
```

The first example is Forth written like Pascal or C. You take the pseudocode that outlines the functions and then transform it into a giant blob of working code.

It is typical of the code I wrote before the big "Aha!" Though I can't call it a mystical experience, I think I finally understand what Charles Moore is driving at as the Forth Way. In Forth, the pseudocode becomes the Forth words. These words are then fleshed out in Forth one-liners. The words describe the problem and the solution.

Leo Brodie, in *Thinking Forth*, does a good job of explaining Moore's philosophy, but prior experience with Pascal or C definitely screws up the attempt to think Forth. If any other language actually helped me to learn Forth, it was assembler. The paradigm of Forth is assembler, not one of the Backus-Naur Algol derivatives.

I've read complaints that Forth is too primitive, too simple, too open. These are precisely the reasons I like Forth. In my mind's eye, I can see the code work. I don't have to pray to the compiler god and hope things work out.

Let Forth be Forth.

Yours truly,
Walter J. Rottenkolber
P.O. Box 1705
Mariposa, California 95338

(Letters and code continue on page 35.)

Optimization Considerations

Charles Curley
Gillette, Wyoming

This paper describes a 68000-based JSR/BSR threaded Forth interpreter/compiler. The compiler compares a variable and a header field, and either assembles a JSR or BSR to a called word, or copies its code in line. The definition of `;` is smart enough to replace a BSR/JSR at the end of a word with a JMP or BRA, as appropriate. Several words which are not traditionally immediate become so, such as `>R` and constants.

The Forth described herein is FastForth, a full 32-bit Forth for the 68000. It is a direct modification of the indirect-threaded Real-Forth. This is, in turn, a direct descendant of fig-Forth. (Remember fig-Forth?) Vocabularies, among other things, retain their original flavor.

For those not familiar with 32-bit Forths, memory operators with the prefix `W` operate on word, or 16-bit, memory locations.

The Implementation

It is conventional wisdom among Forth gurus that smaller is faster, and faster means smaller. The commonly accepted exception to this has been when it comes to subroutine threading vs. indirect threading. Here, the traditional argument has been that the two bytes per call (say, on a PDP-11) is worth the overhead, compared to four bytes per call. This argument is less attractive on an eight-bit processor, such as the 6502, where a subroutine call is three bytes, and the interpreter for the indirect threading is some 14 instructions.

"But, if we crank up the clock speed..." someone said. Probably someone at Intel, or with equal imagination.

Anything one can get by cranking up the clock speed, one can get by both cranking up the clock speed and by using other techniques, such as better compilers. Or better coding. The 68000's rich instruction set and plentiful supply of addressing modes make it ripe for such improvements.

The traditional Forth compiler looks rather like this:

```
: INTERPRET
  BEGIN -FIND
  IF  ( found) STATE @ <
    IF  CFA ,  ELSE  CFA EXECUTE  THEN
  ELSE  HERE NUMBER  DPL @ 1+
    IF  [COMPILE] DLITERAL
    ELSE  DROP [COMPILE] LITERAL  THEN
  THEN  ?STACK  AGAIN     STOP
```

Paleoforthwrights[1] will no doubt recognize this as the fig-Forth compiler. This system is simple, easy to understand, and fast.

It runs a lot faster if parts of it are written in code, of course. With a 32-bit data path and 32-bit code fields, optimization by assembly language re-coding can go hog wild on the 68000. For example, the word `,` (comma) becomes:

```
CODE ,    OFUSER DP AR0 MOV,
      4 # OFUSER DP ADDQ,
      ' ! 2+ *+ BRA, ;C
```

(`OFUSER` is an assembler macro which assembles a displacement from the user area register.)

Even with this scheme, any word called will still occupy four bytes for each call, plus the overhead of `next` and the return code. But even with this overhead, many words in the nucleus become both smaller and faster.

A major step is taken when one moves from indirect-threaded code to subroutine threading. Whole aspects of Forth are affected, often in a very subtle manner. The code interpreter can stay the much the same. However, it now calls another word to assemble its calls:

```
: <BSR>   2-  HERE -  DUP -80  80 WITHIN
  IF  FF AND  6100 OR
  ELSE  6100 W,  THEN  W, ;

: <SUB>
  \ addr --  | compile subroutine to addr
  HERE OVER - -8000  7FFF WITHIN
  IF  <BSR>  ELSE  4EB9 W,  ,  THEN ;

: INTERPRET
  BEGIN -FIND
  IF  ( found) STATE @ <
    IF  <SUB>  ELSE  EXECUTE  THEN
  ELSE  HERE NUMBER  DPL @ 1+
    IF  [COMPILE] DLITERAL
    ELSE  DROP [COMPILE] LITERAL  THEN
  THEN  ?STACK  AGAIN     STOP
```

1. "Forthwright" is a term coined by Al Kreever. The "paleo" prefix is my own perversion. I also use the term "neologist" for someone who creates new words. Forth is, after all, the language for people who like to play with words.

<SUB> is now a lot more than , is, and we have added a lot to the dictionary that wasn't already there. <SUB> calculates whether to use a BSR or JSR instruction, and uses the appropriate one. <BSR> is smart enough to use a short or long relative call, as needed.

By making this change, we must also redefine next. Instead of a three-instruction (six bytes) macro, we now have a one-instruction (two byte) macro. The instruction is, of course, RTS.

Now a reference to a called word may be two bytes, four bytes, or six, depending on how far away the call is. We have made next a lot smaller. In the nucleus, there are no six-byte calls, and quite a lot of two-byte calls. We have reduced the size of the nucleus considerably, and gained speed.

Another innovation is to get rid of the words 0BRANCH and BRANCH, which do the work of controlling flow in conditional branches. These, of course, are replaced with processor-instruction equivalents. 0BRANCH and BRANCH occupy six bytes per call, four for the CFA and two for the displacement if the branch is taken. The processor instructions occupy four or six bytes each, and run much faster.

The relevant code is:

```
: 0BRAN 201B6700
  ( s [+ dr0 .l mov,   2 *+ eq bcc, )
  ,  0 W,   HERE  2- ;
: BRAN   60000000 ( 2 *+ bra, )
  ,  HERE 2- ;
: RESOLVE          HERE OVER -  SWAP  W! ;

: IF
  ?COMP  0BRAN 2 ;    IMMEDIATE
: THEN
  ?COMP  2 ?PAIRS  RESOLVE ;  IMMEDIATE
: ELSE
  2 ?PAIRS  BRAN
  SWAP 2 [COMPILE] THEN  2 ;  IMMEDIATE

: BEGIN ?COMP HERE 1 ;  IMMEDIATE

| : SBKWD  10000 /  HERE ROT SWAP -
    2-   FF AND OR  W, ;
| : LBKWD           HERE ROT SWAP -
    2- FFFF AND OR   , ;
| : BKWD  OVER HERE -  2-  -7F 7F WITHIN
    IF SBKWD ELSE LBKWD THEN ;

: UNTIL 1 ?PAIRS
  201B W,   67000000 BKWD ;      IMMEDIATE
: AGAIN 1 ?PAIRS
            60000000 BKWD ;      IMMEDIATE
: REPEAT
  >R >R [COMPILE] AGAIN
  R> R> 2- [COMPILE] THEN  ;     IMMEDIATE
: WHILE [COMPILE] IF 2+ ;        IMMEDIATE
```

These words, again, occupy more room in the dictionary than their predecessors did, but the code compiled by them is so much smaller and faster that the overhead is worth it.

Because the 68000 has an efficient instruction set, and calls to a word may be as many as six bytes long, it is possible to have calls to words which occupy more room than the word itself. Why not copy the guts of such words in line, and forget the call? We may or may not save space in the dictionary, but we can get rid of the overhead of the call and return instructions. This requires a change in the interpreter:

```
: <BSR>   REL OFF  2-  HERE -
  DUP -80  80 WITHIN
  IF  FF AND  6100 OR
  ELSE  6100 W,  THEN  W, ;

: <SUB>
  \ addr --  | compile subroutine to addr
  HERE  OVER - -8000  7FFF WITHIN
  IF  <BSR>  ELSE  4EB9 W,   ,  THEN ;

: <COMP>
  \ addr --  | subroutine or inline?
  DUP 2- W@  -DUP
  IF LENGTH  @ 1+ <
    IF  HERE OVER 2-  W@  DUP ALLOT  CMOVE
    ELSE  <SUB>  THEN
  ELSE  <SUB>  THEN ;

: INTERPRET
  BEGIN -FIND
  IF  ( found) STATE @ <
    IF  <COMP>  ELSE  EXECUTE  THEN
  ELSE  HERE NUMBER  DPL @ 1+
    IF  [COMPILE] DLITERAL
    ELSE  DROP [COMPILE] LITERAL  THEN
  THEN  ?STACK  AGAIN     STOP
```

The word <COMP> looks at a field in the word's header, the length field. If the contents of the field is zero, no copy is made—a subroutine is called instead. If the length field is less than or equal to the current contents of the user variable LENGTH, an in-line copy of the target word is made, instead of a subroutine call. If the length field is greater than LENGTH, a subroutine call is made in the normal fashion.

The use of a variable to determine the cutoff for copying code allows the programmer to select the best length for such copying. For most uses, this is set to six, so dictionary size is the main consideration. However, it can be set to any value up to 32K, if the user wants to really go for speed. (None of this "We're from Microsoft and we know more about your application than you do" stuff here!)

Even if the length value for compiling the nucleus is set to the reasonable minimum, four, we still gain. DUP, for example, has one two-byte instruction in it. It shows up about 70 times in the FastForth nucleus, for 140 bytes of savings. Thus, even though we have overhead in the nucleus to copy in line, we still come out ahead in nuclear size. Other one-instruction, two-byte words abound, such as DROP.

The compiler must know the value to which it must set the length field. This value is best calculated after the word is fully compiled, so the logical place to do it is in ;. That code looks as follows:

```
: NXT
  HERE LATEST N>C  -  SETLEN  4E75 W,   ;
```

The code to be copied (and hence the length of the word for copying purposes) must exclude the return instruction at the end. So we make the calculation before adding the return.

We may have a problem, however. Are there circumstances under which it is inadvisable to make an in-line copy of a word? Answer, yes. One circumstance is when the source word contains a relative reference, such as a program counter relative offset. Or a BSR instruction, which is often. That is why the word <BSR> sets the user variable REL (for "relocatable") to the off state. Thus, the code called by the word ; which sets the length field in the head, must examine REL.

```
: NXT  REL @
  IF  HERE LATEST N>C  -  SETLEN
  THEN  4E75 W,   ;
```

This code requires that the length field be set to zero, which is done by CREATE. And the relocation indicator must be set to its default state:

```
: :   ?EXEC !CSP  CURRENT @ CONTEXT !
  CREATE  SMUDGE
  REL ON  LATEST N>C  DP !  ] ;
```

The phrase LATEST N>C DP ! is there because CREATE sets up a code field pointing to the code for variables, and this must be overwritten by a : definition. This is a byproduct of the decision to modernize the system by having CREATE produce variables, instead of the ancient fig-Forth practice of having it produce headers for code definitions.

A gotcha of 68000s is that the 68000 is word-aligned for word and long-word memory accesses. That is, either @ must pick up data a byte at a time and assemble the four bytes, or it cannot be used on odd address cells. The latter alternative would be incompatible with other Forths running on other processors, so the former was selected for Real-Forth, the immediate ancestor of FastForth. The result is as follows:

```
CODE @   S [ AR0 .L MOV,
\ avoid byte boundary
   AR0 [+   S  [ .B MOV,         \ problems
   AR0 [+ 1 S &[ .B MOV,
   AR0 [+ 2 S &[ .B MOV,
   AR0 [   3 S &[ .B MOV,   NEXT  ;C
```

Aside from being ugly, the word takes up 16 bytes in memory. It probably will be referred to a lot by subroutine call. However, why not provide both types of memory access? A version of @ requiring word alignment produces a four-byte word:

```
CODE F@          \ @ from even address only
S [ AR0 MOV,  AR0 [ S [ MOV,  NEXT  ;C
```

This word will invariably be copied in line. Furthermore, it will get used a lot: all variables, user variables, and word or long word arrays are word aligned. Thus, careful editing of the nucleus produces a much faster nucleus, using F@ where appropriate. Alas, the nucleus grows—but not much. Because this word will be copied in line, application references to it will produce smaller applications, so the cost is well worth it.

Similar logic produces F! from !:

```
CODE F!        \ store to even address only
S [+ AR0 MOV,  S [+ AR0 [ MOV,  NEXT  ;C
```

Can we squeeze more room out of the nucleus and still accelerate things? Well, it seems a bit absurd for the last instruction in a word (before next) to be a subroutine call. Why not force the call to become a jump? Once execution of the called word ends, the return instruction will force execution back to the word which called the current word. The RTS instruction may then be omitted from the end of the word. This saves us an instruction in the dictionary, and two return stack accesses in execution.

The resulting code gets tricky. There are two circumstances under which this trick is inadvisable: when the last instruction before the return is not a call, and when a forward branch within the word refers to where the RTS would be if there were one. To check for the latter condition, we simply examine a variable maintained by the compiler word THEN. THEN is now defined as:

```
: THEN  HERE LASTTHEN F!
  ?COMP  2 ?PAIRS  RESOLVE ;  IMMEDIATE
```

So we know where the last subroutine call was made, the compiler now maintains a variable, LASTSUB.

```
: <SUB>
  \ addr --  | compile subroutine to addr
  HERE DUP LASTSUB F!  OVER -
  -8000  7FFF WITHIN
  IF  <BSR>  ELSE  4EB9 W,  ,  THEN ;
```

The code to make it work all operates from ;.

```
| : DOBSR
    = IF    60  LASTSUB F@ C!  0
    ELSE  1  THEN ;

| : 6SR   LASTSUB F@  W@ 4EB9
    = IF  4EF9  LASTSUB F@ W!  0
    ELSE  1  THEN ;

| : 2SR   LASTSUB F@  W@ FF00 AND 6100
    DOBSR ;

| : 4SR   LASTSUB F@  W@ 6100
    DOBSR ;
```

```
\ --- fl | 1 indicates failure to change
\            bsr to bra, etc.
| : LAST?  REL F@ 0=  HERE
    LATEST N>C -  LENGTH F@ 1- >  OR
    IF  HERE LASTSUB F@ -
    DUP 2 =
      IF  DROP 2SR
      ELSE    DUP 4 =
        IF  DROP 4SR
        ELSE 6 =
          IF  6SR  ELSE  1  THEN
      THEN  THEN
    ELSE 1 THEN ;

: NXT   REL F@
  IF  HERE LATEST N>C  -  SETLEN  THEN
  4E75 W,  ;

: ;  ?COMP ?CSP  HERE LASTTHEN F@ -
  IF  LAST?
    IF  NXT  THEN
  ELSE  NXT  THEN
  SMUDGE [COMPILE] [ ;   IMMEDIATE
```

The three words 6SR, 4SR, and 2SR each handle the three possible subroutine call instructions. They do this by munging[2] the last subroutine call's opcode into a BRA or JMP opcode, as appropriate. The word LAST? determines whether to call one of these words and, if so, which one. It also returns a flag to indicate whether the final RTS instruction should be added to the word. Also, the whole process is bypassed if there is a forward branch pointing to where the RTS would be.

This has the effect of giving us a free, zero-instruction return, without having to build a custom processor to do it.

Further Optimizations

One can make further optimizations along the same lines. The fig-Forth word LIT goes away entirely, to be replaced by an in-line literal instruction. For large values, the following instruction obtains:

```
<value> # s -[ mov,
```

For smaller values, the MOVQ instruction proves useful:

```
<value> # dr0 movq,  dr0 s -[ mov,
```

LITERAL (still FIGgishly state smart) is redefined as follows:

```
: LITERAL   STATE F@
  IF  DUP -80 7F WITHIN
    IF  FF AND 7000 OR W,  2700 W,
    ELSE 273C W, ,  THEN
  THEN ;  IMMEDIATE
```

Constants can be reworked in a major way. We can produce a word which is not relocatable, and therefore requires a subroutine call for each reference. Instead, we make constants into immediate words (!) which simply produce literals as needed:

```
: CONSTANT   CREATE IMMEDIATE ,
  DOES> F@ [COMPILE] LITERAL ;
```

Similar surgery may be committed on variables.

```
: VARIABLE  CREATE  , IMMEDIATE
  DOES>  LITERAL ;
```

User variables require a more complex operation at compile time, as they must compile an opcode and the offset from the user pointer (maintained in a register on the 68000). In addition, execution at interpretation time is more complex. The resulting word has a hybrid code and high-level ;CODE portion.

```
: USER   CREATE  W, IMMEDIATE  ;CODE
  RP [+ AR0 MOV,  U AC &[ ( ofuser state )
  TST,  NE IF,
    DR0 CLR,  AR0 [ DR0 .W MOV,
    DR0 S -[ MOV,  ]] 41EE W,  W,  2708 W,
    \ ofuser <n> ar0 mov,  ar0 s -[ mov,
    [[  ELSE,  U DR0 MOV,
    AR0 [ DR0 .W ADD,
    DR0 S -[ MOV,   THEN,    NEXT  ;C
```

The first two lines of the ;CODE portion examines the user variable STATE to determine whether the system is compiling or interpreting. If the system is compiling, the next two lines are executed. The offset of the user variable is brought into a register, sign extended, and pushed onto the data stack. Then high-level words are executed to comma in the first opcode, 41ee, the offset (an argument to the first opcode), and then finally the second opcode, 2708. This results in the assembly in line of the following code fragment:

```
ofuser <n> ar0 mov,  ar0 s -[ mov,
```

If the system is not compiling, the actual address of the user variable is calculated by adding the offset to the contents of the user register. The results are pushed onto the data stack.

In most other languages, a lot of hand coding would be done to make these compact definitions possible. Fortunately, Real-Forth and FastForth both provide both an assembler and a disassembler, so code definitions can be prototyped and the object values determined rapidly. Such tools are essential for language development.

Another area of optimization is to move the indices and limits of loops into the 68000's data registers. This will produce faster and probably more compact code. (Isn't it nice to have an adequate supply of registers?) Rather arbitrarily, data registers five and six were selected to hold the index and the limit, respectively. (DO) pushes these onto the return stack, and the loop ending operators pop them off.

2. PDP-11 hacker slang for stomping on the object code directly. Attributed to Mung the Merciless.

```
ASSEMBLER BEGIN,
  2DUP  >R >R   2 # AR0 ADDQ,
  RP [+ DR6 MOV,  RP [+ DR5 MOV,
  AR0 [ JMP,

CODE (LOOP)
RP [+ AR0 MOV,   1 # DR5 ADDQ,
  LABEL LP2      DR5 DR6 CMP,
                 R> R> GT UNTIL,
  LABEL LP5      AR0 [ AR0 .W ADD,
                 AR0 [ JMP,   ;C
FIXED
>R >R

CODE (+LOOP)
RP [+ AR0 MOV,
  S [+ DR0 MOV,  DR0 DR5 ADD,
  DR0 TST,   PL LP2 *+ BCC,
  DR5 DR6 CMP,   LT LP5 *+ BCC,
  R> R> AGAIN,     ;C   FIXED

\       dr5: index
dr6: limit
CODE (DO)
RP [ AR0 MOV,  DR5 RP [ MOV,
  DR6 RP -[ MOV,
  S [+ DR5 MOV,   S [+ DR6 MOV,
  AR0 [ JMP, ;C    FIXED
```

Nuclear gurus are reminded that this is still a fig-Forth nucleus, and there are differences in how the loop operators work between fig-Forth and later standards.

(DO) operates by pushing two items from registers onto the return stack. In order to do this, it first pops the return address into AR0. The loop registers are pushed, and the new index and limit are popped from the data stack. An RTS is emulated by jumping indirect through AR0, which holds the return address.

(LOOP) works by comparing the two data registers. In all cases, the return address is first popped into AR0. If the loop is not exhausted, the offset to return to the beginning of the loop is added to AR0, and a jump indirect through AR0 is executed. If the loop is exhausted, execution branches to the code fragment ahead of (LOOP). There, the return address is adjusted to skip over the offset. The two data registers are popped from the return stack, and execution is resumed with the usual indirect jump through AR0.

Since we have moved loop indices and limits from their traditional places on the return stack, index and limit operators must also change. I must be recoded:

```
CODE I  DR5 S -[ MOV,   NEXT   ;C
```

R[3] can no longer be aliased to I, and must now be a separate word.

J and other words which access nested loop limits and indices must also be recoded. J now looks like the old I.

We also need a way for the Real-Forth hackers to twiddle the loop index while in a loop. For example, (EXPECT) plays with the loop index when it sees a backspace character. This is handled by writing the new word I!, which allows sufficiently unstructured code to be an eyesore.

```
CODE I!  S [+ DR5 MOV,   NEXT   ;C
\ use to play w/ index

: (EXPECT) OVER + OVER
\ add for Atari/IBM PC keyboard
     DO KEY DUP 14 +ORIGIN W@ =
     OVER 16 +ORIGIN W@ = OR
       IF DROP 08 OVER I = DUP I 2- + I! -
       ELSE DUP 0D =
         IF LEAVE DROP BL 0
         ELSE  DUP THEN
       I C!  0 I 1+ C! THEN
     EMIT ( DROP)
     LOOP  DROP ;
```

The Implications

Such drastic surgery on a nucleus has implications elsewhere in the nucleus, for application coding and for utility code such as decompilers. Even one's conceptual view of Forth is affected.

The most profound shock, especially for those of us accustomed to fig-Forth-styled dictionaries, is that the concepts of the parameter field and code field merge and become one. (This is not, however, an approach toward a Grand Unified Field Theory.) The most disconcerting thing for a fig-Forth user is that ' and its relatives can no longer return the parameter field. It may or may not be the same as the code field; however, the code field will always exist. So ' and its FastForth brethren now return the code field address.

This changes the family of words used to maneuver in the header of a word to the point where they had to be renamed. They now take their names from the field address they expect and the one they return. For example, to navigate from the code field to the name field, one uses C>N. To go the other direction, N>C.

This name change has the benefit of aiding conversion of code from Real-Forth (or other Forths) to FastForth.

A word in this family is C>P, used to get from the code field to the parameter field, if there is one. This word must skip over the instruction at the code field, which will be one of three possible subroutine calls. This it does by detecting which instruction is there. It works as follows:

```
: C>P
  ( cfa --- pfa  | find parameter field )
  DUP  W@ 6100 = IF   4+  ELSE
  DUP  W@ 4EB9 = IF  6 +  ELSE  2+
  THEN THEN ;
```

Occasionally one has need to go back the other way. That is stranger:

```
: P>C
\ pfa --- cfa | jump from pseudonfa (pfa)
\ of a voc to its code field
```

3. R@ to you mezo- or neoforthwrights.

```
DUP  2- W@  FF00 AND  6100 =
IF  2-
ELSE DUP  4- W@  6100 =
  IF  4-
  ELSE DUP  6 - W@  4EB9 =
    IF  6 -  ELSE  ABORT" bad link"
THEN THEN THEN ;
```

P>C makes guesses about which instruction was used, and where it would be if it had been used. This word is not in the nucleus, because it is used so rarely. It was originally constructed to allow vocabulary-traversing code to print out the names of the vocabularies in the system as it traversed the linked list of vocabularies.

Another conceptual change will hit the Forth nucleus guru or the person who does much assembly language programming under Forth. This is that the IP and W registers have moved. The Forth instruction pointer is now the processor's instruction pointer—sometimes. W is now the first cell on the processor stack. Usually.

For an example of how this works, let's look at an intermediate definition of VARIABLE. This was implemented to act exactly like the indirect-threaded version of VARIABLE, and requires a call to each variable. It has since been replaced by the version given above. Note that the code field is set by the word CREATE.

```
: VARIABLE  CREATE , ;

  ASSEMBLER  HERE  *VARIABLE*  !
  RP [+ S -[ MOV,  NEXT

: (CREATE)
  FIRST HERE 0A0 + U< 2 ?ERROR
  ?ALIGN -FIND
  IF DROP C>N ID.   4 MESSAGE SPACE THEN
  HERE  DUP C@ WIDTH F@ MIN
  1+ =CELLS ALLOT
  DUP 80 TOGGLE HERE 1- 80 TOGGLE
  LATEST , CURRENT F@ F!  0 W,
  { *VARIABLE* F@ } LITERAL <SUB> ;
```

Since the length field of a variable is never changed from its initial zero, all references to variables are by subroutine. This subroutine call places the return address on the stack. The first instruction in the variable is another subroutine call, to the working code routine for variables. This instruction also places a return address on the return stack. But the second return address points to the variable's allocated storage area, not to code. So all the working code has to do is pop the address off the return stack and push it to the data stack. The next instruction, the RTS, resumes execution at the code which called the variable.

The ability to copy in-line code into a word means that the locations of return stack items get rather fuzzy. An item is going to be somewhere on the return stack, but where depends on whether the calling word copied the target word in line or not. For example, a subroutine version of R would have to reach over the return address to get the value on the return stack to be copied. An in-line version would not have to skip the return address.

An in-line version of R is only one instruction, two bytes. It makes sense to copy it in line wherever possible. But it isn't always possible: some of us use the return stack to store things at interpretation time:

```
BASE @ >R HEX  ...  R> BASE !
```

The implementor could be bloody-minded about the whole thing and tell you not to do things like that. Or he could have written a set of state-stupid words for use inside compiled words, and another set of state-stupid words for use outside of compilation, and he could have expected you, the user, to remember the difference.

But Allah is merciful. Instead, we have three state-smart immediate words, R, >R, and R>. For example:

```
\ rp [ s -[ mov, => 2717
CODE R  OFUSER STATE TST,
  NE IF, 2717 #L S -[ MOV,
  'NF W, *+ BRA,  THEN,
  4 RP &[  S -[ MOV,
  NEXT  ;C       IMMEDIATE
```

They all work on the same model. If the system is compiling, the appropriate opcode is assembled in line with W,. Otherwise, a subroutine version is executed.

This also means that LENGTH may never be set so that R is called by subroutine. That is, it may never be less than two.

Execution arrays have also mutated under FastForth. With indirect threading, all references to words in the array were the same length. Thus, indexing into the array was easy: multiply the index by the size of the reference, add it to the base address of the array, fetch the value there, and execute it. In 32-bit Real-Forth, EXEC is defined as follows (except that it is done in code):

```
: EXEC   4*  R> + @ EXECUTE ;
```

The new version is a bit more elaborate. The old EXEC mutates into:

```
CODE <EXEC>
\ index ---   | index into execution array
  S [+ DR0 MOV,   2 # DR0 ASL,
  RP [+ AR0 MOV,  DR0 AR0 ADD,
  AR0 [ AR0 MOV,  AR0 [ JMP,  ;C
```

And a new compiler directive is added:

```
: EXEC   COMPILE <EXEC>   BEGIN -FIND
  IF  ( found) STATE F@ <
    IF  ,  ELSE  EXECUTE  THEN
  ELSE  0 ?ERROR
  THEN  ?STACK  STATE F@ 0= UNTIL  ;
  IMMEDIATE
```

EXEC simply compiles a series of CFAs in line, until it finds that compilation has been turned off, usually by the word STOP.

Other Improvements

There is much work one can to do to optimize FastForth. Most of these suggestions have been done, at least experimentally. Their implementations and implications will be left as an exercise for the student. There will be a quiz.

Forward-referring branches can be made smart enough to make two- or four-byte branches, if one cares to write the code to move the intervening code appropriately at branch resolution time.

Since the processor has a variety of conditional branch instructions, why not make the Forth conditional branches reflect this? The traditional Forth typically compiles two words: one performs a test, the other does the branch. Instead, why not make the branch instruction also do the test? For example, the phrase `0= IF` might become two in-line instructions at compile time, instead of three or more.

We have seen how to move the indices and limits for loops into registers. Why not save more time at run time, and force `(DO)` and `(LOOP)` (and their ilk) to always be copied in line? This will require changes in the way `DO` and `LOOP` operate at compile time.

A major improvement can be made in any Forth by changing the header structure. The traditional fig-Forth header structure places the link field after the name field in memory. This requires dictionary searches to traverse each name field to go to the next word in the dictionary. By placing the link field before the name field, the traverse loop is replaced with a single instruction. Since compilation consists largely of dictionary searches, compilation is greatly speeded.

Interim Results

There are plenty of optimizations yet to make in FastForth. In spite of this, one may make some preliminary assessments. The results are not all in, but they are definitely promising. For a quick and dirty benchmark, I looked to the Eight Queens problem, as coded by LeVan, *Forth Dimensions* II/1, and modified by Wilson M. Federici (GEnie e-mail address W.FEDERICI). As I am also using an Atari ST, my results compare directly with Mr. Federici's. However, to speed things up, I made the arrays byte arrays, which eliminates a two-place shift, and replaced `F@` with `C@`. I also found that the greatest speed for any given version was achieved with FastForth's `LENGTH` set to 16.

To Federici's results, I add the final five entries:

F32:	8.90 sec.
ForST with CALLS:	7.23 sec.
ForST with MACROS:	3.77 sec.
Real-Forth 1.3 (ITC):	
LW cells & 2*2*	7.60 sec.
LW cells & cell*	7.06 sec.
byte cells	5.65 sec.
FastForth 2.0 (J/BTC):	
LW cells & 2* 2*	4.87 sec.
byte cells	3.04 sec.

Compilation times improved. For example, compiling the target compiler and then target compiling the FastForth nucleus, approximately 13 kilobytes in size, takes about 120 seconds under Real-Forth. Under FastForth, this improves to under 70 seconds. (As the Atari ST has a real processor, there is enough room to hold the source for all this in Forth's local memory, so speed of disk access is excluded from consideration.)

Conclusions

Properly done, conversion of a 68000 32-bit Forth from indirect-threaded code to subroutine-threaded code will be rewarding in both speed improvements and in application- and nucleus-size improvements. The speed improvements were expected when the conversion process was started, as was the smaller nucleus. The improved application size was a pleasant surprise.

But the key point is this: however snappy compilers or other tools may help (or hinder), they are no substitute for competent programming or competent software design. They are especially no substitute for good optimization. And those are all still arts.

Availability

Persons wishing to experiment with FastForth may implement these techniques on their own target compilers for personal use and experimentation. Those who wish to run the complete FastForth package may obtain a beta site copy for the Atari ST from the author. The author will also discuss ports to other 680x0 machines and ports to other processors with interested parties.

Charles Curley is a long-time Forth nuclear guru who lives in Wyoming. He earns his living as a paralegal so he can afford necessities like 68000-based Forth systems and luxuries like food and rent. He may be reached at P.O. Box 2071, Gillette, Wyoming 82717-2071.

MAKE YOUR SMALL COMPUTER
THINK BIG

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

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

MMSFORTH System Disk from $179.95
Modular pricing — Integrate with System Disk only what you need:

FORTHWRITE - Wordprocessor	$99.95
DATAHANDLER - Database	$59.95
DATAHANDLER-PLUS - Database	$99.95
FORTHCOM - for Communications	$49.95
GENERAL LEDGER - Accounting System	$250.00

MMSFORTH

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

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

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

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

Modular pricing — Integrate with System Disk only what you need:

EXPERT-2 - Expert System Development	$69.95
FORTHCOM - Flexible data transfer	$49.95
UTILITIES - Graphics, 8087 support and other facilities.	

and a little more!

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

Call for free brochure, technical info or pricing details.

Forth-Based Message Service

Olaf Meding
McFarland, Wisconsin

Charles Moore, the inventor of Forth, brought a newspaper clipping dated "March Forth" to his keynote address at this year's SIG-Forth conference in Kansas City, Missouri. Whether we "march" or "boldly go," this paper describes how Amtelco's EVE (Electronic Video Exchange) has became the predominant, largest, and most sophisticated messaging system for the telephone answering service (TAS) industry. EVE is used not only in every major urban area in the United States and Canada, but in Australia and throughout Thailand. EVE has gained 70% of the TAS market and is still growing. Forth—which is used exclusively—is a key ingredient of the success story you are about to read.

System Description

Before focusing on the software architecture, I would like to briefly describe EVE. EVE is the center of a telephone answering service (see Figure One). Thirty-two operators take messages by phone from callers, and later deliver those messages to the clients. In addition to normal business hours, messages are frequently taken on weekends, holidays, and evenings. Typical clients include doctors, small companies, business managers, and travelling sales personnel. Typical EVE owners include answering services, mail order houses, executive suites, and paging services. At Amtelco, we use an in-house EVE station to handle all field service calls.

EVE provides all necessary functions needed by a modern TAS: telephony functions, paperless message handling, client database (10,000 accounts), maintenance, and client billing. EVE routes a few thousand telephone trunks to thirty-two operators. The operators type and retrieve messages into the database. EVE is capable of handling well over 12,000 phones calls per day.

Messages can be stored, retrieved, delivered, archived, or purged. There are a variety of ways in which messages can be delivered, i.e., verbally by the operators, remotely printed via modems or FAXes, and paged through wireless paging terminals. Clients can also use a personal computer to log into EVE to get their messages.

An operator can log in as a supervisor to perform system maintenance. Such tasks include maintaining client account information, retrieving statistics used for measuring operator performance, and client billing.

In addition, field service personnel can dial into EVE via a modem to monitor all aspects of the system, such as locating hardware, software, and database problems. A number of software routines are available to field service. For example, one repairs a broken database. Owners of EVE stations throughout the U.S.A. and Canada have formed a very dynamic users group, the National Association of EVE owners (NAEO).

EVE was born ten years ago in 1982. Since then, 17 programmers have worked on EVE software for a total of 38 man years. To give the reader an idea of the size of the EVE application, it is comprised of about 100,000 lines of source code. The average size of the EVE Software Works programming team is four full-time Forth programmers. It is worth noting that none of the programmers employed had any previous Forth experience. They were hired as full-time programmers, not consultants. With very few exceptions, all the programmers were able to master Forth and EVE code, and became very successful in their careers.

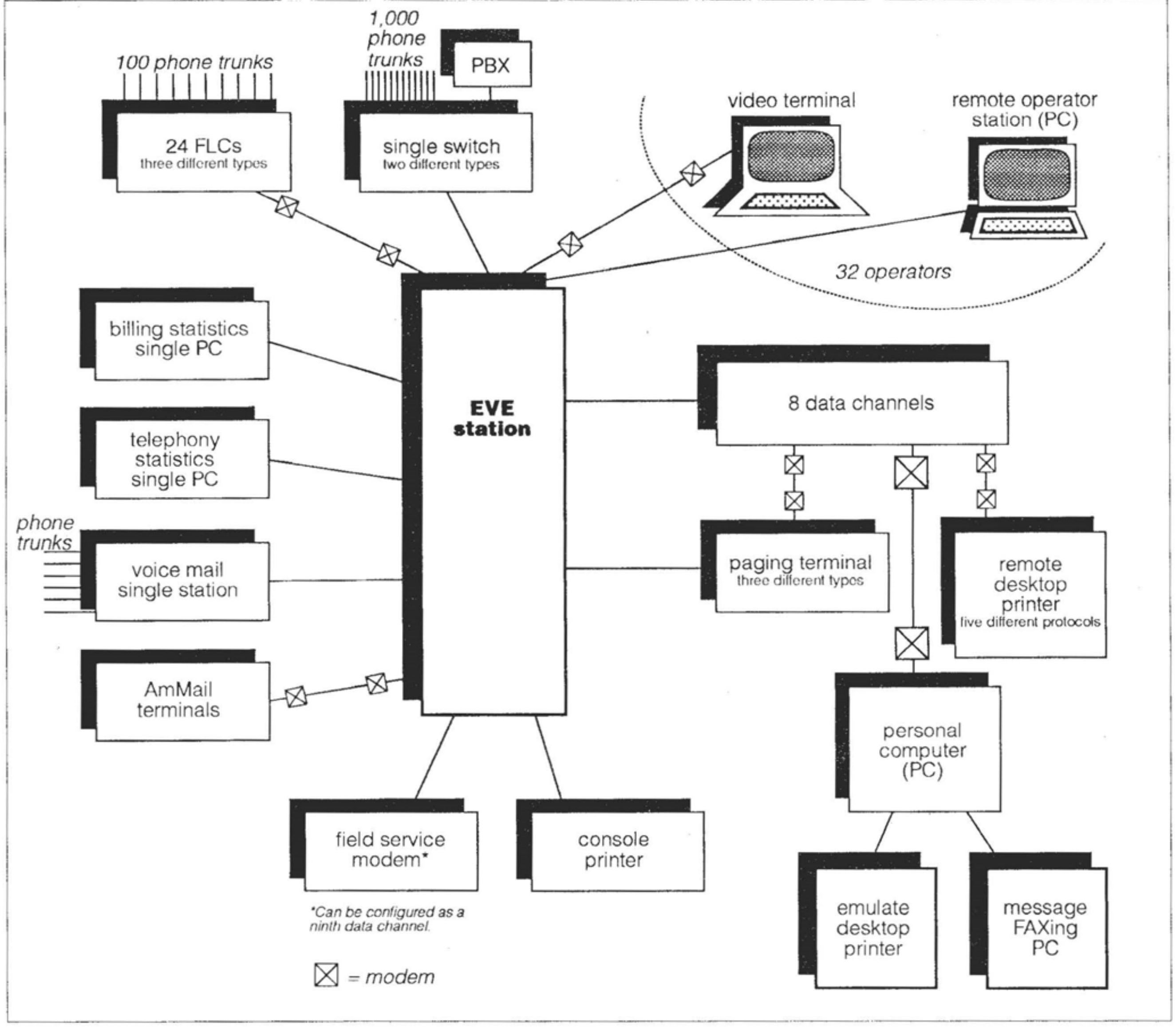
Forth's Contribution

Forth is much more than a computer language. Forth is a complete programming environment, and even more it is a philosophy. The concept of simplicity is what makes Forth so effective and powerful. Interestingly, long after Forth was invented, the same concept of simplicity was introduced to microprocessors through RISC (reduced instruction set computer) technology, which increases a CPU's throughput.

Forth uses words. Words are the equivalent of a subroutine in C or Pascal. One problem with subroutines is that they tend to get very large (over a page of source code). A Forth word is composed on average of five to nine other Forth words. Because each word has a name, the code becomes highly readable in itself. Even though the C or Pascal equivalent of a Forth word is a subroutine, a Forth word acts more like a constant. A small section of code is given a name; in large applications, it is important to use words with descriptive names rather than a magical sequence of instructions.

The freedom and flexibility Forth extends to the programmer is reflected by the versatility and wide range of possible EVE station configurations (Figure One). EVE takes full advantage of many powerful Forth features, such as the extremely efficient round-robin multi-tasker and an extraordinarily efficient database. Rather than depending on the compiler vendor for enhancements needed to further de-

Figure One.



velop software (as is the case with most non-Forth programming languages), we tailored our Forth environment (polyForth by Forth, Inc.) to match the underlying hardware for maximum code efficiency and execution speed. The power of Forth should be evident by the fact that the entire EVE software, consisting of up to 84 independent software tasks (except for the asynchronous I/O boards), is run by a single 10 MHz 68000 Motorola microprocessor.

Forth has made it possible to consistently and quickly respond to the demands of our customers and the TAS industry. Forth's combined power of programming environment, operating system, database manager, multi-tasking, complete availability of source code, and striking philosophy of simplicity are the reasons why a system first developed in 1982 is still number one in the market. Forth truly shines in rapid prototyping during software development and debugging, which in turn dramatically decreases the time-to-market of innovative new EVE software editions.

A medium-size EVE station costs well over $100,000. Forth protects the owner's investment by making it possible to continuously expand the software without making the hardware investment obsolete.

All software, including firmware and a large number of device drivers, is written in Forth. Amtelco was one of the first users of SCSI hard drives, but we could not afford a $25,000 SCSI bus analyzer. Forth enabled us to write SCSI device drivers without expensive bus-analyzing hardware. I remember the surprised look of a visiting drive engineer when we presented him with a bug in his SCSI drive. He found it hard to believe that we used Forth without a SCSI bus analyzer to write device drivers.

A Challenge—Remote Operator Stations

A recent addition to EVE, remote operator stations consist of a personal computer rather than a video terminal. Remote operators use a pair of high-speed, asynchronous modems to connect to the EVE station. Two benefits of the remote operator are telecommuting (working at home) and allowing sales staff to set up new client accounts at the point of sale with a live demonstration of how their calls will be handled.

Another benefit of remote operators is that supervisors can monitor the system at home and take action only when needed. Finally, a multi-tasking remote operator can be connected to more than one EVE station simultaneously.

The development of the remote operator station was one of our most challenging projects. It was very difficult to design a highly interactive application asynchronously. We had a number of problems initially because the remote station would get stuck waiting for a response from EVE. For example, we had to develop our own communications protocol because the connection between the modem and the host would often lose data, and the modem did not detect line breaks fast enough. Forth is an ideal environment for this type of application because of its powerful multi-tasking capability.

To decrease hardware costs, management decided that remote operator station personal computers should operate without a hard disk. Again, Forth proved to be the most suitable language. Forth made it possible to design very efficient code that could be booted from floppy disk without the need for software overlays.

The remote operator was an essential part of the Thailand project. Forth enabled us to rapidly write a VGA graphics display driver for the Thai character set, and to streamline EVE's message-paging capabilities for our Thai customers. We heard that even the King of Thailand has paging accounts on the Bangkok EVE station!

Developing Powerful Software Tools

Forth made it possible to write our own software tools. A good example, and one of the first tools we developed, is still the most powerful. It is called `COMPARING`. It compares a range of Forth blocks and highlights the differences in the code on the screen or printer (variations are printed in bold simply by printing them twice).

With multiple programmers (at one time we had seven) working on a major software edition, comparing is used to print all changes. These comparisons are given to an integrator for integration into the final release. The printouts are also used to verify all code changes at the end of the development cycle. This works especially well if a different programmer (usually the integrator) verifies the changes.

`ZEUS` is another unique and powerful debugging task written to aid in the development of the remote operator station. It is a background task running on the remote station. A programmer can log onto a customer's EVE station (Figure Two) via modem. The programmer then loads a utility on EVE which allows sending actual Forth commands through a second pair of modems to the remote operator station. The Forth command is executed on the remote operator station, and the result is sent back through EVE to the programmer's terminal via the four modems. The remote operator is undisturbed in its operation and continues to take messages while, in the background, a programmer is debugging the system. The name Zeus is a reference to the power of this debugging tool.

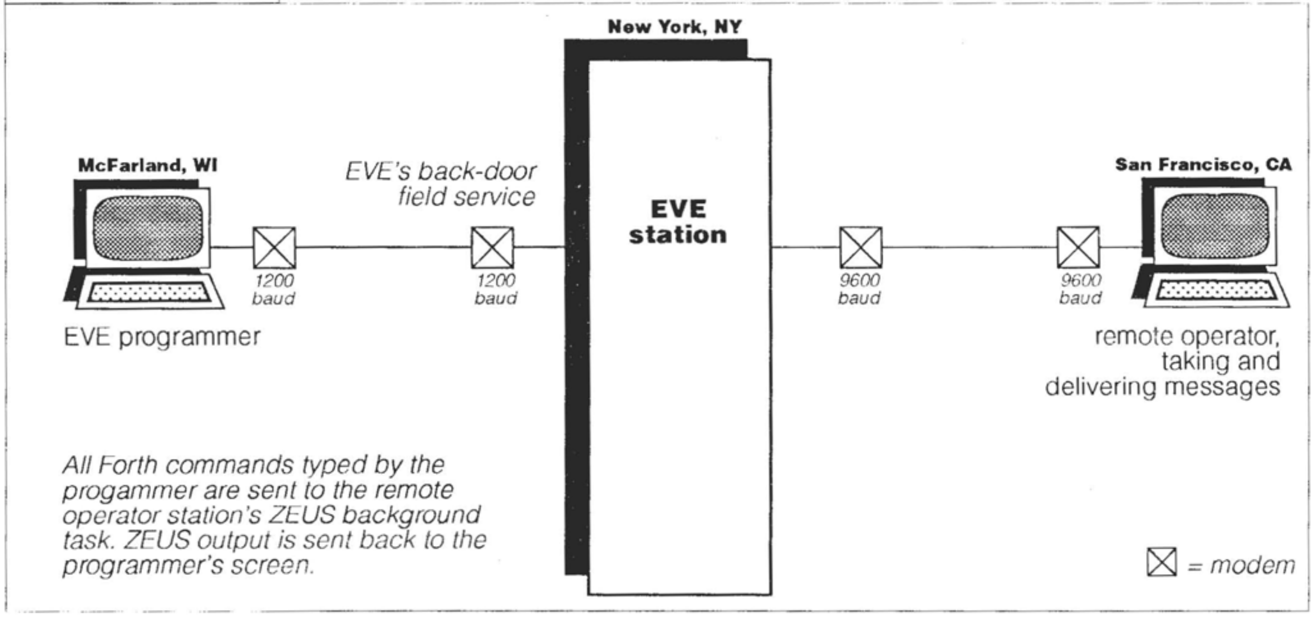
The power of Forth is limited only by the imagination of the programmer, and Zeus is a good example of this. Once the idea for Zeus was born, Forth was the perfect environment to realize the idea and concept.

The debugging task on EVE is named `TRON`. Those who saw the movie know why.

Conclusion

Forth is neither a low- nor a high-level programming environment—it is both and more. The highly interactive Forth environment greatly stimulates the process of converting human ideas and thoughts into machine code. For this reason, I believe that Forth programmers spend most of their time solving problems rather than trying to work around restrictions imposed by other, non-Forth programming methods. This highly productive process of writing Forth software builds an even higher level of confidence in the programmer, which in turn significantly reduces the number of errors (bugs). The programmer's confidence in error-free code is a key ingredient of successful "Forth on a grand scale" projects.

Figure Two.


New York, NY
McFarland, WI
EVE's back-door field service
EVE station
San Francisco, CA
1200 baud
1200 baud
9600 baud
9600 baud
EVE programmer
remote operator, taking and delivering messages
All Forth commands typed by the progammer are sent to the remote operator station's ZEUS background task. ZEUS output is sent back to the programmer's screen.
⊠ = modem

Graphics and Floating Point in Real-Time Action

Dr. Mark Smiley
Baltimore, Maryland

A Zenith-150 spewing fractal dragons on the screen introduced me to Forth. Articles in *Byte* and *Dr. Dobbs Journal* increased my suspicion that Forth and fractals would wed well.

At the time, I was using Fortran on a mainframe to draw pictures of the fractals whose mathematical properties I was studying. I'd drive an hour to reach a site where I had to program on a terminal in one building, then walk a good distance to another building, where I frequently had to tap on the window to wake up the operator in order to get the plotter output from my program. The idea of owning my own computer, one with video graphics, appealed to me. I purchased a Z-150 and set about learning Forth. Thus began my relationship with Forth and graphics on MS-DOS machines.

An early bad experience with a Forth vendor, and a desire to have access to all the source code, led me to public-domain Forths. Yet none of them contained graphics routines. I resolved to create enough routines to enable my studies. This effort eventually resulted in the F-PC graphics package currently distributed through the Forth Interest Group's software library.

I wanted a vocabulary that would make it easy to express my ideas—for that I required floating point.

Forth's interactive nature lends itself to work with graphics. My labors in this area led to a number of applications. Some I was paid to develop, others helped with my dissertation, but many I wrote just for fun. This article discusses the genesis of the graphics routines in the light of Juliam, an application I sell that grew symbiotically alongside the graphics routines.

Juliam

Juliam craves a rich Forth environment. It requires a wide range of graphics and floating-point routines, as well as a professional menu system for constructing a friendly interface. In addition, the graphics routines allocate large (64K) buffers outside the Forth system, and support a variety of graphics modes: CGA, EGA, VGA, and many SVGA boards. Together with the more than 800K of graphics routines, Juliam comprises well over a megabyte of code.

Juliam uses a variety of algorithms to draw both Julia and Mandelbrot sets. To get an idea of how these sets are defined, consider the map: f(z) = z2 + C, where z and C are complex numbers. For example, suppose C=0 and z=2. Then f(2)=4, f(4)=16, and f(16)=256. Thus 4, 16, and 256 are iterates of 2; they are examples of the output obtained from applying the map f(z) over and over again, each time plugging the output back into f(z). Note that the iterates of 2 increase without bound. In other words, they go towards infinity. On the other hand, the iterates of 1/2 approach 0. Furthermore, the iterates of -1 go neither to 0 nor to infinity. This point lies on the boundary between the points that go to infinity and those that don't. Thus -1 represents an element of the Julia set for the map f(z) = z2. Indeed, the Julia set for this map is just the circle of radius 1 centered at the origin.

In general, Julia sets take on far more intricate patterns than mere circles. Roughly, the Julia set of the map f(z) = z2 + C consists of those points, z, that lie on the boundary between the set of points whose iterates go to infinity and those that don't. (More precisely, to mathematicians, it is the closure of the set of repelling periodic points.) Hence Julia sets reside in z-space. On the other hand, the Mandelbrot set is the set of all values C for which the Julia set is connected, so the Mandelbrot set sits in C-space. For a far fuller discussion of Julia and Mandelbrot sets, see [2].

The list below presents some of the features of Juliam version 5.11. These items may give you an idea of some of the routines I struggled to implement.

1) Real-time interactive graphics—watch the Julia set change as you alter the parameters.
2) Milnor/Thurston algorithm for the Mandelbrot set—more detail than the common, forward-iteration algorithm.
3) Move a crosshair about on the Mandelbrot set and draw the corresponding Julia set.
4) Move a resizable rectangle on the screen and zoom in on the image.
5) Save images to disk, complete with all pertinent parameters.

6) Create and watch mini-movies.
7) Support for a variety of graphics cards: CGA, EGA, VGA, and many SVGA cards.

Graphics

The first public-domain Forth I tried was a version of MVP-Forth. I used a DOS interrupt to put the computer in graphics mode, and another to plot a single point. Both used `INTCALL ( ax bx cx dx interrupt -- ax )`. Later, I migrated to Laxen and Perry's F83 Forth, then the variants F83S, F83SX, F83Y, Wil Baden's F83X, and finally Tom Zimmer's FF, F88 and F-PC. On MS-DOS machines, all BIOS video is handled through interrupt $10 (a $ means hexadecimal in F-PC). `INT$10` below represents a simplified version of `INTCALL` for use with F-PC, though the current routines no longer use anything as general (or as slow).

```
{
CODE  INT$10       ( ax bx cx dx -- )
\ call interrupt $10
    POP DX    POP CX    POP BX   POP AX
    PUSH BP   INT $10   POP BP
NEXT END-CODE

: VGA320   ( -- )
\ enter VGA 320x200 256-color mode
       $13 0 0 0 INT$10    ;

: B.DOT.OLD   ( x y color -- )
              ( column row color -- )
     $C00 + -ROT 0 -ROT  INT$10 ;
}
```

Of course it is faster to avoid the stack thrashing of `B.DOT.OLD`. Here's the current BIOS version of `B.DOT`.

```
{
CODE B.DOT    ( x y color -- )
              ( column row color -- )
    POP AX       MOV AH, # $0C
    \ AH=function, AL=color
    XOR BX, BX     \ page
    POP DX         \ y-coordinate
    POP CX         \ x-coordinate
    PUSH BP        \ preserve BP register
    INT $10        \ call BIOS
    POP BP         \ restore BP
NEXT   END-CODE
}
```

It is even faster to bypass the BIOS and write directly into video memory. Here's an example of a direct screen-writing version of `DOT` for VGA mode $13 (320x200 with 256 colors). Compare it with the slower `B.DOT`. (One disadvantage of direct routines is that nearly every graphics mode requires a different version.)

```
{
\ vga mode $13, 1 pel/byte (320x200)
code v320.dot   ( x y color -- )
    POP DI             \ color in di
    pop cx  pop dx     \ dx = x, cx = y
    push es            \ save ES
    mov ax, # $a000    \ video seg in memory
    mov es, ax   \ write directly to memory
    xor bx, bx         \ base of buffer
    mov bx, dx         \ cx = row, dx = col
    mov ax, # 320      \ pixels per row
    cwd
    mul cx             \ rows to our dot
    add bx, ax         \ bx = offset
    mov ax, di         \ di = color (0-255)
    mov es: 0 [bx], al     \ write pixel
    pop es                 \ restore ES
    next  c;
}
```

Even back when I worked in the MVP dialect, I taught a class in Forth. One of my colleagues, Johnny Graves, audited my course and wrote the high-level, line-drawing routine `DLINE`. His code still turns up in public-domain Forth code today. Later, I cleaned it up and converted it to assembler for speed. A year after Johnny, a talented student named Tim Smith wrote some line-drawing routines in assembler for F83. As with `VGA320.DOT` above, Tim's code bypassed the BIOS and wrote directly to video memory. As a result, it ran significantly faster than the high-level `DLINE`.

At first, I only had access to CGA systems. Later, when EGA and VGA became available, Mike Sperl (via the now-defunct East Coast Forth Board) helped me port and optimize the code further.

Juliam's accurate Mandelbrot set algorithm requires filled disks, so I added these, too, along with the aspect ratios necessary to achieve circles. This algorithm was first discussed in a paper, which languished long unpublished, by Milnor and Thurston, each of whom has won the Fields Medal—the equivalent of the Nobel prize for mathematics. Their algorithm is an example of a distance estimator method (DEM). It uses a sophisticated technique to estimate the distance from a given point to the Mandelbrot set. Then it draws a disk centered at that point which contains no points of the Mandelbrot set, and calls itself recursively on four of the disk's boundary points. Thus the algorithm fills in all that is not the Mandelbrot set.

To see the difference between the DEM algorithm and the traditional one that so many other programs use, compare Figure One (traditional) and Figure Two (DEM). Both depict the Mandelbrot set.

To speed up the filled disk routines, I added some optimized routines for horizontal lines. Later, I wanted to include a feature in Juliam that would allow a user to move a crosshair about on the Mandelbrot set to select a value of C to generate a new Julia set. For maximum speed, I implemented optimized horizontal and vertical lines that XORed onto the screen. These lines also helped in a feature that lets users move a resizable rectangle around on an image to select a region for zooming.

Some images take several hours to generate, so I needed the ability to save them to disk. The key word for implementing a crude (and slow) version of this graphics screen saving is BIOS-READ-DOT. Given the coordinates of a pixel, it returns the pixel color.

```
{
CODE  BIOS-READ-DOT    ( x y -- color )
    POP DX             \ y-coordinate
    POP CX             \ x-coordinate
    XOR BX, BX         \ page 0
    MOV AX, # $D00     \ BIOS service number
    PUSH BP
    INT $10
    POP BP
    AND AX, # $00FF  \ clear AH to 00, so
            \ that AX=AL is just the color
    1PUSH   END-CODE
}
```

In the current graphics package, all but some new SVGA modes now have routines that can save an entire image at once, rather than calling a routine like BIOS-READ-DOT. Some routines put the image in memory, for quickly saving and restoring a screen during program execution; other routines save to disk, for permanence.

It's funny how the first words that must be written to develop a graphics package—words to enter graphics modes—appear near the end of a completed graphics package. As the graphics package migrated and grew, I found that more and more words needed to be deferred, so each graphics resolution required its own word to set all these values. Thus, words like VGA320 that controlled graphics modes, moved from the top of the graphics package to the bottom. Furthermore, experience led to separating these words into a part like (VGA320) to set all the parameters, and another part GRMODE that actually entered graphics modes. This separation makes it possible to generate computations based on the graphics mode, without actually entering the mode.

To make things more complicated, some systems demand BIOS graphics, so each graphics resolution required two words: one to set deferred words like LINE to use BIOS graphics, and the other to set the words to use direct screen-writing graphics. Typical examples are (VGA320.D) to set the direct routines of VGA mode $13, and (VGA320.B) for the BIOS routines.

The package contains words to facilitate switching between BIOS and direct techniques. DIRECT_GRAPHICS switches all graphics modes to use the direct routines, while BIOS_GRAPHICS switches them all to use the BIOS ones. The BIOS routines retain more compatibility, but significantly less speed, than the direct routines.

Floating Point

My studies required not only graphics, but floating point. True, many calculations could be performed with integer arithmetic. After all, pixels have integer screen coordinates. But I wanted a vocabulary that would make it easy to express my ideas, and for that I required floating point.

At my first FIG conference in 1985, I met Roland Koluvec, who kindly provided me with a hardware floating-point package that a friend of his (Stephen Pollack) had written for F83S. Later, I gave a copy of it to the Silicon Valley FIG library. Since F83 worked with blocks, I put both the floating-point and graphics routines in one large file that clearly indicated the authorship of its various parts. Later, others extracted the

Figure One. Traditional.

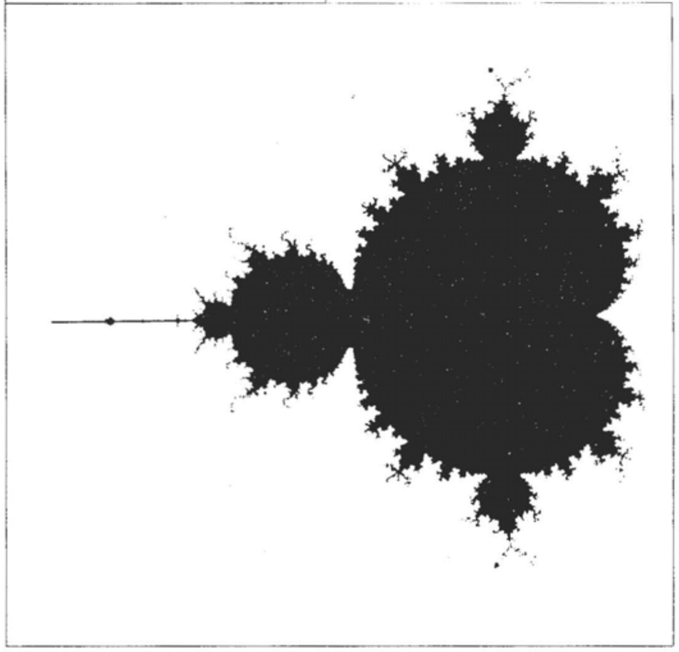
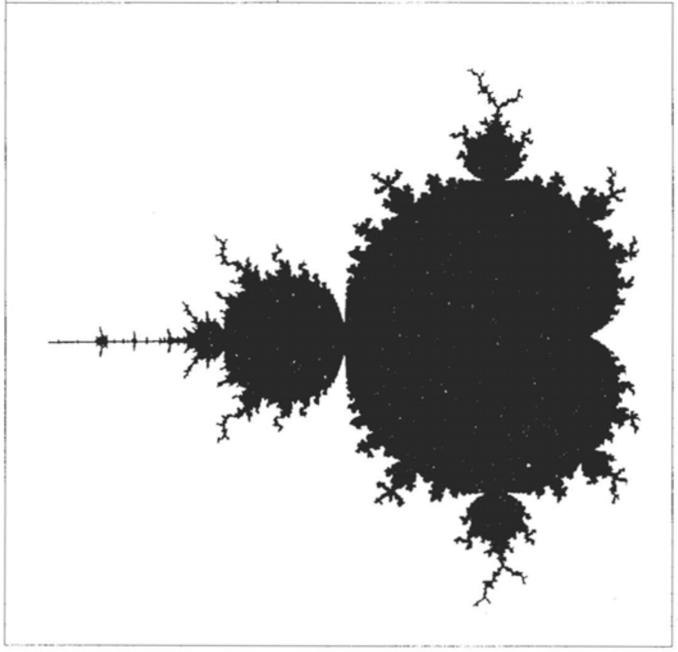


Figure Two. DEM.



floating-point part and converted it to HFLOAT.SEQ in Tom Zimmer's Forths. In the process, some people came to believe (erroneously) that I wrote the floating-point package. This is not the case: I merely added a few extensions. Unfortunately, the package had bugs in it and, as far as I know, some of those bugs persist today in the current incarnation, though I no longer have access to an 80x87 chip to test this hypothesis.

Drawing fractals seems to be a good test for floating-point routines. My experiments have helped me catch a variety of errors in Bob Smith's SFLOAT.SEQ—bugs which Bob typically squashed within hours. Today, SFLOAT is an excellent package with no known bugs.

VPSFLOAT is another high-quality, software floating-point package for F-PC. It has a little more accuracy than SFLOAT, but it is also slower. My work uncovered a minor bug in it, too, which Jack Brown rapidly quelled. One day, he has promised to convert a good hardware floating point to F-PC, too. I look forward to that day.

After I had a floating point, I developed the first version of Juliam, which only drew Julia sets in those days, and so was called JULIA.BLK in F83, and later JULIA.SEQ in F-PC. You can get a version of the latter in the F-PC graphics package.

Using the Graphics Package

Applications must first select the graphics resolution by setting the deferred word (RES), which prepares all the parameters for the graphics mode without actually entering it. For example:

```
' (VGA320) IS (RES)
```

Then the program should utilize either GRMODE or SET-RES to enter the graphics mode. Here's an example:

```
{
: test1
  ['] (VGA320) IS (RES)
   \ set current graphics mode, by setting
   \ the value of (RES).
   set-res    \ enter current graphics mode
   200 100 15 dot
   \ plot a point with coordinates
   \ (200,100) and color 15 (white).
   key drop              \ wait for a key
   text                  \ return to text mode
;
}
```

Better yet, use CHOOSE-RES (or the SVGA version, RESMENU) to allow the user to select the graphics mode from a menu of available modes, as in the next example.

```
{
: test2
   choose-res       \ set the value of (RES)
   set-res    \ enter current graphics mode
   200 100 15 dot
   \ plot a point with coordinates
   \ (200,100) and color 15 (white).
   key drop          \ wait for a key
   text              \ return to text mode
;
}
```

I have only scratched the surface of the graphics package and all that is possible with it. It includes things like turtle graphics, VGA sprites, graphics text fonts for positioning text with pixel-level control, world coordinates, automatic function plotting, VGA palettes, a flood fill, 3D to 2D transformations, and much more.

The graphics package is an ongoing project. I continually imagine new features that would improve the package; the ones I really need, I implement. As you can see, many people have contributed to the development of the graphics and floating-point routines. If you are interested in adding a new routine or improving an existing one, feel free to contact me and I will work with you to integrate it into the graphics package. Here are some ideas: the ability to save and read GIF, PCX, PCL, or EPS files; direct support for more SVGA cards; palette rotation; fast filled polygons; 3D hidden surfaces with shading; and ray tracing. I am also interested in any applications you write that employ aspects of the graphics package. Drop me a line describing them. Better yet, send me a copy.

References

1. Mandelbrot, Benoit B., *The Fractal Geometry of Nature*, W. H. Freeman and Co., 2nd edition, New York (1982).

2. Peitgen, Hans Otto, *The Beauty of Fractals*, Springer-Verlag.

(Code begins on next page.)

In 7th grade, a friend taught Mark Smiley the essentials of Fortran while they sat in a Dairy Queen. In 1979, he received a B.S. magna cum laude in mathematics from Denison University, where he first experienced computer graphics via BASIC on a plotter for a PDP-11. In 1983, he attained an M.S. in mathematics from the University of North Carolina. His thesis, under Dr. Sheldon Newhouse, lies in the realm of dynamical systems/ergodic theory: *Relations between Hausdorff Dimension, Lyapunov Exponents, and Entropy.*

He spent the next four years teaching at Auburn University in Alabama, where he wrote Fortran programs to draw pictures related to his research.. He used the public-domain version of MVP-Forth to teach Forth for the first time. He and some of his students wrote the first bits of a graphics package, which he later ported to F83, F83X, FF, F88, and F-PC.

In 1987, Mark married Cathy (who has an M. S. in mathematics). In 1990, he achieved a Ph.D. in mathematics from Auburn University. His dissertation is titled *Metric Dimensions of Fractals.* He used F-PC to generate a number of images for the dissertation.

He currently is an Assistant Professor in the Department of Mathematics and Computer Science at Goucher College, near Baltimore, Maryland. He has taught Forth at there, but teaches BASIC more frequently—which led him to write the textbook *Learning QuickBASIC Through VGA Graphics*, (Kendall/Hunt Pub. Co., 1992).

"Graphics & Floating Point" code.

```
\ Graphics Variables
variable vres                    \ vertical resolution
variable hres                    \ horizontal resolution
variable color                   \ in the range 0 to #colors-1
variable #colors                 \ number of different colors
variable vid.mode                \ video mode
variable palette                 \ the default palette
variable vchip                   \ video card chip set
variable buf.size                \ size of the graphics buffer
variable vid.seg                 \ video memory segment
$FA00  value  bit_plane.size     \ number of bytes to read/write
variable bytes/row               \ used by direct graphics routines
64000. POINTER BUF.SEG           \ screen save buffer
\ F-PC uses POINTER to allocate memory from outside Forth.

\ Plotting Pixels
defer dot           ( x y color -- )    \ plot a pixel
defer color-dot     ( x y -- )          \ uses value of COLOR for color
defer clip-dot      ( x y -- )          \ uses value of COLOR for color
defer xdot          ( x y color -- )    \ XORs a pixel
defer cdot          ( color x y -- )    \ plot a pixel
defer read-dot      ( x y -- color )    \ return the color of a pixel

\ All lines use the variable COLOR as the color.
defer line          ( x1 y1 x2 y2 -- )  \ draw a line
defer xline         ( x1 y1 x2 y2 -- )  \ XOR a line
defer hline         ( x1 x2 y -- )      \ draw a fast horizontal line
defer xhline        ( x1 x2 y -- )      \ XOR a fast horizontal line
defer vline         ( y1 y2 x1 -- )     \ draws a vertical line
defer xvline        ( y1 y2 x1 -- )     \ XORs a vertical line
defer nline         ( x y -- )          \ draws from current point to (x,y)
}
NLINE draws a line from the current point to the point on TOS.
Use MOVETO ( x y -- ) to set the current point before invoking NLINE the
first time.

The following words save and restore graphics screens from video memory
save either to buffer or to disk, depending on the graphics mode
{
defer SaveVid
}
Save the current graphics screen to a temporary location. This location may
be either file or memory, depending on the graphics mode. (To save the
screen permanently, the graphics package provides BSAVE and "BSAVE.)
{
defer RestVid        \ Restore screen from temporary location.
defer TEXT           \ enter text mode (i.e. get out of graphics mode)
defer (bsave)        \ used by BSAVE and "BSAVE to save to disk
defer (brecall)      \ used by BRECALL to view an image on disk

DEFER (RES)          \ set graphics parameters

\ resolutions: values for (RES)
DEFER (MED)          DEFER (HIGH)         DEFER (EGA)
DEFER (VGA640)       DEFER (VGA320)

}
!VMODE stores values for various resolution-dependent variables.
This version of !VMODE assures that the true HRES is stored in
OLD_HRES whether the image is square or not.
{
: !VMODE     ( vidmode hres vres #colors vidseg bufsize vchip -- )
    VChip !                \ chip set
    BUF.SIZE ! VID.SEG ! #COLORS !
    VRES !
    OLD_HRES !             \ save old horizontal resolution
    SQ_IMAGE? @               \ if image is square,
    IF                        \ make the resolution square
       VRES @
    ELSE
       OLD_HRES @
    THEN
    HRES !  VID.MODE !
    Set-Aspect                \ set the ASPECT, based on the current resolution.
    Set-Pals                  \ set deferred palette words for current res
    Set-White   \ set various values for use in whiting and filling the screen
;

              ( vidmode hres vres #colors vidseg  bufsize vchip )
: (VGA320.D)  ( -- )  19  320  200  256     $A000   $FA00   VGA
    !VMODE
    $FA00 !> bit_plane.size
    hres @ bytes/row !
    1 =: #PLANES
    ['] v320.dot is dot
    ['] v320.xdot is xdot
    ['] v320.cdot is cdot
    ['] v320.color-dot is color-dot
    ['] v320.clip-dot is clip-dot
    ['] bios-read-dot is read-dot
    ['] line320 is line
    ['] xline320 is xline
    ['] hline320 is hline
    ['] xhline320 is xhline
    ['] 256vline is vline
    ['] x256vline is xvline
    ['] D.NLINE IS NLINE
    ['] vid>buf IS SaveVID
    ['] buf>vid IS RestVID
    ['] (cga_brecall) is (brecall)
    ['] (cga_bsave) is (bsave)          \ for "BSAVE
    ['] BUF_BSAVE IS BSAVE
    ['] WHITE-VGA IS WHITE-SCREEN
    ['] FILL-VGA IS FILL-SCREEN
    ['] (CLEAR-SCREEN.D) IS (CLEAR-SCREEN)
;
' (VGA320.D)  IS  (VGA320)

: (>BIOS)
    ['] B.dot is dot
    ['] B.xdot is xdot
    ['] B.cdot is cdot
    ['] B.color-dot is color-dot
    ['] B.clip-dot is clip-dot
    ['] bios-read-dot is read-dot
    ['] B.line is line
    ['] B.HOR_line is hline
    ['] B.VER_line is vline
    ['] B.xline is xline
    ['] B.xhline is xhline
    ['] B.xvline is xvline
    ['] B.NLINE IS NLINE
    ['] SLOW-WHITE-SCREEN  IS  WHITE-SCREEN
    ['] SLOW-FILL-SCREEN IS FILL-SCREEN
    ['] Set-Res IS (CLEAR-SCREEN)       \ clear screen using INT $10
;
}
```

(Code continues on next page.)

```
>VGA_BIOS sets various deferred words for saving images in VGA and SVGA modes.
{
: >VGA_BIOS  ( -- )
    (>BIOS)
    Set_Write_Size          \ set the Write_Size, #Rows/Write and
                            \ #Writes/Im for use in saving images
    ['] SaveVGA_BIOS IS SaveVID
    ['] RestVGA_BIOS IS RestVID
    ['] (file>vga) IS (brecall)
    ['] (vga>file) IS (bsave)
    ['] COPY_IMAGE IS BSAVE
;

             ( vidmode hres vres #colors vidseg  bufsize vchip )
: (VGA320.B) ( -- )  19  320  200  256     $A000   $FA00   3
    !VMODE
    >VGA_BIOS
    ;
\ ' (VGA320.B)  IS  (VGA320)

}
DIRECT_GRAPHICS makes graphics commands write directly to the screen.
It is deferred, so that other modes may be patched in later.
{
DEFER DIRECT_GRAPHICS
: (DIRECT_GRAPHICS)
    ['] (MED.D)   IS   (MED)
    ['] (HIGH.D)   IS   (HIGH)
    ['] (EGA.D)   IS   (EGA)
    ['] (VGA320.D)   IS   (VGA320)
    ['] (VGA640.D)   IS   (VGA640)
    (RES)    ;              \ makes changes take effect
' (DIRECT_GRAPHICS)   IS   DIRECT_GRAPHICS
DIRECT_GRAPHICS

DEFER BIOS_GRAPHICS
: (BIOS_GRAPHICS)
    ['] (MED.B)   IS   (MED)
    ['] (HIGH.B)   IS   (HIGH)
    ['] (EGA.B)   IS   (EGA)
    ['] (VGA320.B)   IS   (VGA320)
    ['] (VGA640.B)   IS   (VGA640)
    (RES)    ;              \ makes changes take effect
' (BIOS_GRAPHICS) IS  BIOS_GRAPHICS

CODE MODE   ( n -- )        \ enter graphics mode n
    POP AX
    INT $10
    NEXT C;

}
The words STATOFF and SLOW below are necessary in F-PC to avoid the text
writing directly to the screen in graphics modes, which would result in
unintelligible garbage.
{
: GRMODE                    \ enter the current graphics resolution
    STATOFF   SLOW
    VID.MODE @   MODE   ;


: SET-RES                   \ set appropriate values and enter the
    (RES)                   \ current graphics mode
    GRMODE
;
```

FIG
MAIL ORDER FORM

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

"Were Sure You Wanted To Know..."

***Forth Dimensions*, Article Reference** 151 - $4 0#
★ An index of Forth articles, by keyword, from *Forth Dimensions* Volumes 1–13 (1978–92).

FORML, Article Reference 152 - $4 0#
★ An index of Forth articles by keyword, author, and date from the FORML Conference Proceedings (1980–91).

FORTH DIMENSIONS BACK VOLUMES

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

Volume 1 Forth Dimensions (1979–80) 101 - $15 1#
Last 50 Introduction to FIG, threaded code, TO variables. fig-Forth.

Volume 3 Forth Dimensions (1981–82) 103 - $15 1#
Last 5 Forth-79 Standard, Stacks, HEX, database, music, memory management, high-level interrupts, string stack, BASIC compiler, recursion, 8080 assembler.

Volume 6 Forth Dimensions (1984–85) 106 - $15 2#
Last 100 Interactive editors, anonymous variables, list handling, integer solutions, control structures, debugging techniques, recursion, semaphores, simple I/O words, Quicksort, high-level packet communications, China FORML.

Volume 7 Forth Dimensions (1985–86) 107 - $20 2#
Last 100 Generic sort, Forth spreadsheet, control structures, pseudo-interrupts, number editing, Atari Forth, pretty printing, code modules, universal stack word, polynomial evaluation, F83 strings.

Volume 8 Forth Dimensions (1986–87) 108 - $20 2#
Last 100 Interrupt-driven serial input, data-base functions, TI 99/A, XMODEM, on-line documentation, dual-CFAs, random numbers, arrays, file query, Batcher's sort, screenless Forth, classes in Forth, Bresenham line-drawing algorithm, unsigned division, DOS file I/O.

Volume 9 Forth Dimensions (1987–88) 109 - $20 2#
Last 100 Fractal landscapes, stack error checking, perpetual date routines, headless compiler, execution security, ANS-Forth meeting, computer-aided instruction, local variables, transcendental functions, education, relocatable Forth for 68000.

Volume 10 Forth Dimensions (1988–89) 110 - $20 2#
Last 50 dBase file access, string handling, local variables, data structures, object-oriented Forth, linear automata, stand-alone applications, 8250 drivers, serial data compression.

Volume 11 Forth Dimensions (1989–90) 111 - $20 2#
Last 100 Local variables, graphic filling algorithms, 80286 extended memory, expert systems, quaternion rotation calculation, multiprocessor Forth, double-entry bookkeeping, binary table search, phase-angle differential analyzer, sort contest.

Volume 12 Forth Dimensions (1990–91) 112 - $20 2#
Last 100 Floored division, stack variables, embedded control, Atari Forth, optimizing compiler, dynamic memory allocation, smart RAM, extended-precision math, interrupt handling, neural nets, Soviet Forth, arrays, metacompilation.

FORML CONFERENCE PROCEEDINGS

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

1980 FORML PROCEEDINGS 310 - $30 2# **Last 10**
Address binding, dynamic memory allocation, local variables, concurrency, binary absolute & relocatable loader, LISP, how to manage Forth projects, n-level file system, documenting Forth, Forth structures, Forth strings. *231 pgs*

1981 FORML PROCEEDINGS 311 - $45 4# **Last 50**
CODE-less Forth machine, quadruple-precision arithmetic, overlays, executable vocabulary stack, data typing in Forth, vectored data structures, using Forth in a classroom, pyramid files, BASIC, LOGO, automatic cueing language for multimedia, NEXOS—a ROM-based multitasking operating system. *655 pgs*

1982 FORML PROCEEDINGS 312 - $30 4# **Last 100**
Rockwell Forth processor, virtual execution, 32-bit Forth, ONLY for vocabularies, non-IMMEDIATE looping words, number-input wordset, I/O vectoring, recursive data structures, programmable-logic compiler. *295 pgs*

1983 FORML PROCEEDINGS 313 - $30 2# **Last 100**
Non-Von Neuman machines, Forth instruction set, Chinese Forth, F83, compiler & interpreter co-routines, log & exponential function, rational arithmetic, transcendental functions in variable-precision Forth, portable file-system interface, Forth coding conventions, expert systems. *352 pgs*

1984 FORML PROCEEDINGS 314 - $30 2# **Last 100**
Forth expert systems, consequent-reasoning inference engine, Zen floating point, portable graphics wordset, 32-bit Forth, HP71B Forth, NEON—object-oriented programming, decompiler design, arrays and stack variables. *378 pgs*

1986 FORML PROCEEDINGS 316 - $30 2# **Last 100**
Threading techniques, Prolog, VLSI Forth microprocessor, natural-language interface, expert system shell, inference engine, multiple-inheritance system, automatic programming environment. *323 pgs*

1987 FORML PROCEEDINGS 317 - $40 3# **Last 30**
Includes papers from '87 euroFORML Conference. 32-bit Forth, neural networks, control structures, AI, optimizing compilers, hypertext, field and record structures, CAD command language, object-oriented lists, trainable neural nets, expert systems. *463 pgs*

1988 FORML PROCEEDINGS 318 - $40 2# **Last 100**
Includes 1988 Australian FORML, Human interfaces, simple robotics kernel, MODUL Forth, parallel processing, programmable controllers, Prolog, simulations, language topics, hardware, Wil's workings & Ting's philosophy, Forth hardware applications, ANS Forth session, future of Forth in AI applications. *310 pgs*

1989 FORML PROCEEDINGS 319 - $40 3# **Last 50**
Includes papers from '89 euroFORML. Pascal to Forth, extensible optimizer for compiling, 3D measurement with object-oriented Forth, CRC polynomials, F-PC, Harris C cross-compiler, modular approach to robotic control, RTX recompiler for on-line maintenance, modules, trainable neural nets. *433 pgs*

1990 FORML PROCEEDINGS 320 - $40 3# **Last 50**
Forth in industry, communications monitor, 6805 development, 3-key keyboard, documentation techniques, object-oriented programming, simplest Forth decompiler, error recovery, stack operations, process control event management, control structure analysis, systems design course, group theory using Forth. *441 pgs*

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

Fax your orders 510-535-1295

1991 FORML PROCEEDINGS 321 - $50 3#
Includes 1991 FORML, Asilomar, euroFORML '91, Czechoslovakia and 1991 China FORML, Shanghai. Differential File Comparison, LINDA on a Simulated Network, QS2: RISCing it all, A threaded Microprogram Machine, Forth in Networking, Forth in the Soviet Union, FOSM: A FOrth String Matcher, VGA Graphics and 3-D Animation, Forth and TSR, Forth CAE System, Applying Forth to Electric Discharge Machining, MCS96-FORTH Single Chip Computer. *500 pgs*

BOOKS ABOUT FORTH

ALL ABOUT FORTH, 3rd ed., June 1990, Glen B. Haydon 201 - $90 4#
Annotated glossary of most Forth words in common usage, including Forth-79, Forth-83, F-PC, MVP-Forth. Implementation examples in high-level Forth and/or 8086/88 assembler. Useful commentary given for each entry. *504 pgs*

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

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

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

FORTH: A TEXT AND REFERENCE 219 - $31 2#
Mahlon G. Kelly & Nicholas Spies
A textbook approach to Forth, with comprehensive references to MMS-FORTH and the '79 and '83 Forth standards. *487 pgs*

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

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

FORTH ENCYCLOPEDIA, Mitch Derick & Linda Baker 220 - $30 2#
A detailed look at each fig-Forth instruction. *327 pgs*

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

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

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

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

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

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

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

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

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

STACK COMPUTERS, THE NEW WAVE 244 - $62 2#
Philip J. Koopman, Jr. (hardcover only)
Presents an alternative to Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC) by showing the strengths and weaknesses of stack machines (hard-cover only).

STARTING FORTH (2nd ed.), Leo Brodie 245 - $29 2#
In this edition of *Starting Forth*—the most popular and complete introduction to Forth—syntax has been expanded to include the Forth-83 Standard. *346 pgs*

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

ACM - SIGFORTH

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

Volume 1 Spring 1989, Summer 1989, #3, #4 911 - $24 2#
F-PC, glossary utility, euroForth, SIGForth '89 Workshop summary (real-time software engineering), Intel 80x8x. Metacompiler in cmForth, Forth exception handler, string case statement for UF/Forth. 1802 simulator, tutorial on multiple threaded vocabularies. Stack frames, duals: an alternative to variables, PocketForth.

Volume 2 #1, #2, #3, #4 912 - $24 2#
ACM SIGForth Industry Survey, abstracts 1990 Rochester conf., RTX-2000. BNF Parser, abstracts 1990 Rochester conf., F-PC Teach. Tethered Forth model, abstracts 1990 SIGForth conf. Target-meta-cross-: an engineer's viewpoint, single-instruction computer.

Volume 3, #1 Summer '91 913a - $6 1#
Co-routines and recursion for tree balancing, convenient number handling.

Volume 3, #2 Fall '91 913b - $6 1#
Postscript Issue, What is Postscript?, Forth in Postscript, Review: PS-Tutor.

1989 SIGForth Workshop Proceedings 931 - $20 1#
Software engineering, multitasking, interrupt-driven systems, object-oriented Forth, error recovery and control, virtual memory support, signal processing. *127 pgs*

1990-91 SIGForth Workshop Proceedings 932 - $20 1#
Teaching computer algebra, stack-based hardware, reconfig-urable processors, real-time operating systems, embedded control, marketing Forth, development systems, in-flight monitoring, multi-processors, neural nets, security control, user interface, algorithms. *134 pgs*

DISKS: Contributions from the Forth Community

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

Prices: Each item below comes on one or more disks, indicated in parentheses after the item number. The price is $6 per disk or $25 for any five disks. 1 to 20 disks = 1 #.

FLOAT4th.BLK V1.4 Robert L. Smith — C001 - (1)
Software floating-point for fig-, poly-, 79-Std., 83-Std. Forths. IEEE short 32-bit, four standard functions, square root and log. **IBM**.

Games in Forth — C002 - (1)
Misc. games, Go, TETRA, Life... Source. **IBM**

A Forth Spreadsheet, Craig Lindley — C003 - (1)
This model spreadsheet first appeared in *Forth Dimensions* VII, 1-2. Those issues contain docs & source. **IBM**

Automatic Structure Charts, Kim Harris — C004 - (1)
Tools for analysis of large Forth programs, first presented at FORML conference. Full source; docs incl. in 1985 FORML Proceedings. **IBM**

A Simple Inference Engine, Martin Tracy — C005 - (1)
Based on inf. engine in Winston & Horn's book on LISP, takes you from pattern variables to complete unification algorithm, with running commentary on Forth philosophy & style. Incl. source. **IBM**

The Math Box, Nathaniel Grossman — C006 - (1)
Routines by foremost math author in Forth. Extended double-precision arithmetic, complete 32-bit fixed-point math, & auto-ranging text. Incl. graphics. Utilities for rapid polynomial evaluation, continued fractions & Monte Carlo factorization. Incl. source & docs. **IBM**

AstroForth & AstroOKO Demos, I.R. Agumirsian — C007 - (1)
AstroForth is the 83-Std. Russian version of Forth. Incl. window interface, full-screen editor, dynamic assembler & a great demo. AstroOKO, an astronavigation system in AstroForth, calculates sky position of several objects from different earth positions. Demos only. **IBM**

Forth List Handler, Martin Tracy — C008 - (1)
List primitives extend Forth to provide a flexible, high-speed environment for AI. Incl. ELISA and Winston & Horn's micro-LISP as examples. Incl. source & docs. **IBM**

8051 Embedded Forth, William Payne — C050 - (4)
8051 ROMmable Forth operating system. 8086-to-8051 target compiler. Incl. source. Docs are in the book *Embedded Controller Forth for the 8051 Family*. **IBM**

68HC11 Collection — C060 - (2)
Collection of Forths, Tools and Floating Point routines for the 68HC11 controller. **IBM**

F83 V2.01, Mike Perry & Henry Laxen — C100 - (1)
The newest version, ported to a variety of machines. Editor, assembler, decompiler, metacompiler. Source and shadow screens. Manual available separately (items 217 & 235). Base for other F83 applications. **IBM, 83.**

F-PC V3.53, Tom Zimmer — C200 - (5)
A full Forth system with pull-down menus, sequential files, editor, forward assembler, metacompiler, floating point. Complete source and help files. Manual for V3.5 available separately (items 350 & 351). Base for other F-PC applications. Req. hard disk. **IBM, 83.**

F-PC TEACH V3.5, Lessons 0–7 Jack Brown — C201a - (2)
Forth classroom on disk. First seven lessons on learning Forth, from Jack Brown of B.C. Institute of Technology. **IBM, F-PC.**

VP-Planner Float for F-PC, V1.01 Jack Brown — C202 - (1)
Software floating-point engine behind the VP-Planner spreadsheet. 80-bit (temporary-real) routines with transcendental functions, number I/O support, vectors to support numeric co-processor overlay & user NAN checking. **IBM, F-PC.**

F-PC Graphics V4.6, Mark Smiley — C203a - (1) **NEW VERSION**
The latest versions of new graphics routines, including CGA, EGA, and VGA suppport, with numerous improvements over earlier versions created or supported by Mark Smiley. **IBM DSDD, F-PC.**

PocketForth V6.1, Chris Heilman — C300 - (1) **NEW VERSION**
Smallest complete Forth for the Mac. Access to all Mac functions, Events, files, graphics, floating point, macros, create standalone applications and DAs. Based on fig & *Starting Forth*. Incl. source and manual. **MAC**, System 7.01 Compatable.

Kevo V0.9b4, Antero Taivalsaari — C360 - (1) **NEW**
Complete Forth-like object Forth for the Mac. Object-Prototype access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Kernel source not included, extensive demo files, manual. **MAC**, System 7.01 Compatable.

Yerkes Forth V3.6 — C350 - (2)
Complete object-oriented Forth for the Mac. Object access to all Mac functions, files, graphics, floating point, macros, create standalone applications. Incl. source, tutorial, assembler & manual. **MAC**, System 7.01 Compatable.

JLISP V1.0, Nick Didkovsky — C401 - (1)
LISP interpreter invoked from Amiga JForth. The nucleus of the interpreter is the result of Martin Tracy's work. Extended to allow the LISP interpreter to link to and execute JForth words. It can communicate with JForth's ODE (Object-Development Environment). **AMIGA, 83.**

Pygmy V1.4, Frank Sergeant — C500 - (1) **NEW VERSION**
A lean, fast Forth with full source code. Incl. full-screen editor, assembler and metacompiler. Up to 15 files open at a time. **IBM.**

KForth, Guy Kelly — C600 - (3)
A full Forth system with windows, mouse, drawing and modem packages. Incl. source & docs. **IBM, 83.**

ForST, John Redmond — C700 - (1)
Forth for the Atari ST. Incl. source & docs. **Atari ST.**

Mops V2.2, Michael Hore — C710 - (2)
Close cousin to Yerkes and Neon. Very fast, compiles subroutine-threaded & native code. Object oriented. Uses F-P co-processor if present. Full access to Mac toolbox & system. Supports System 7 (e.g., AppleEvents). Incl. assembler, docs & source. **MAC**

BBL & Abundance, Roedy Green — C800 - (4)
BBL public-domain, 32-bit Forth with extensive support of DOS, meticulously optimized for execution speed. Abundance is a public-domain database language written in BBL. Req. hard disk. Incl. source & docs. **IBM HD, hard disk reqquired**

WE HAVE CHANGED THE WAY YOU CALCULATE YOUR ORDERS.

1) We have leveled the pricing for FIG items to all members.

2) We have removed the cost of shipping from the price of the items.

3) We have given you a better choice of shipping methods and rates.

Back issues of *Forth Dimensions* and FORML Conference Proceedings are going out of Print!!

fig-FORTH ASSEMBLY LANGUAGE SOURCE

Listings of fig-Forth for specific CPUs and machines with compiler security and variable-length names (see *Installation Manual* below): - $15 1#

OUT OF PRINT

6502	514 - Septem[illegible]	[illegible]	519 - March 81
6809	[illegible]-June 80	**Apple II**	521 - August 81
8080	[illegible] - September 79		

fig-FORTH INSTALLATION MAN[illegible] 501 - $15 1#
Glossary model edit[illegible] [illegible] you purchase this manua[illegible] [illegible]cha[illegible]ny of the source code listings above. *61 pg[illegible]*

OUT OF PRINT

SYSTEMS GUIDE TO fig-FORTH 308 - $25 1#
C. H. Ting (2nd ed., 198[illegible])
How's and why's of the [illegible] Forth Model by Bill Ragsdale, internal structure of fig-Forth system.

OUT OF PRINT

MISCELLANEOUS

T-SHIRT "May the Forth Be With You" 601 - $12 1#
(Specify size: Small, Medium, Large, Extra-Large on order form)
White design on a dark blue shirt.

POSTER (*Oct., 1980 BYTE* cover) **Last 10** 602 - $5 1#

FORTH-83 HANDY REFERENCE CARD 683 - free

FORTH-83 STANDARD 305 - $15 1#
Authoritative description of Forth-83 Standard. For reference, not instruction. *83 pgs*

BIBLIOGRAPHY OF FORTH REFERENCES 340 - $18 2#
(3rd ed., January 1987)
Over 1900 references to Forth articles throughout computer literature. *104pgs*

MORE ON FORTH ENGINES

Volume 10 January 1989 810 - $15 1#
RTX reprints from 1988 Rochester Forth Conference, object-oriented cmForth, lesser Forth engines. *87 pgs*

Volume 11 July 1989 811 - $15 1#
RTX supplement to *Footsteps in an Empty Valley*, SC32, 32-bit Forth engine, RTX interrupts utility. *93 pgs*

Volume 12 April 1990 812 - $15 1#
ShBoom Chip architecture and instructions, Neural Computing Module NCM3232, pigForth, binary radix sort on 80286, 68010, and RTX2000. *87 pgs*

Volume 13 October 1990 813 - $15 1#
PALs of the RTX2000 Mini-BEE, EBForth, AZForth, RTX-2101, 8086 eForth, 8051 eForth. *107 pgs*

Volume 14 814 - $15 1#
RTX Pocket-Scope, eForth for muP20, ShBoom, eForth for CP/M & Z80, XMODEM for eForth. *116 pgs*

Volume 15 815 - $15 1#
Moore: New CAD System for Chip Design, A portrait of the P20; Rible: QS1 Forth Processor, QS2, RISCing it all; P20 eForth Software Simulator/Debugger. *94 pgs*

Volume 16 816 - $15 1#
OK-CAD System, MuP20, eForth System Words, 386 eForth, 80386 Protected Mode Operation, FRP 1600 - 16Bit Real Time Processor. *104 pgs*

DR. DOBB'S JOURNAL

Annual Forth issue, includes code for various Forth applications.

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

FORTH INTEREST GROUP

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

Name		Phone	
Company		Fax	
Street		eMail	
City			
State/Prov.		Zip	
Country			

U.S. Domestic Postage Rates	Surface	2 day Priority	
	$1.00/#	$1.50/#	
International Postage Rates	Surface	AIR MAIL	
	All /#	1-4 #s/#	>4 #s/#
Canada, Mexico	$1.00	$2.00	$1.30
Other Western Hemisphere	$1.00	$3.25	$2.25
Europe	$1.00	$6.00	$4.50
Other International	$1.00	$8.00	$6.00

Item #	Title	Qty.	Unit Price	Total	#

☐ CHECK ENCLOSED (Payable to: FIG)
☐ VISA ☐ MasterCard
Card Number
Signature
Expiration Date
MEMBERSHIP →

Sub-Total		
10% Member Discount, Member #	( )	#s times rate
**Sales Tax on Sub-Total (CA only)		
Postage: Rate ________ x #s		
*Membership in the Forth Interest Group ☐New ☐Renewal $40/46/52		

*MEMBERSHIP IN THE FORTH INTEREST GROUP

The Forth Interest Group (FIG) is a world-wide, non-profit, member-supported organization with over 1,500 members and 40 chapters. Your membership includes a subscription to the bi-monthly magazine *Forth Dimensions*. FIG also offers its members an on-line data base, a large selection of Forth literature and other services. Cost is $40 per year for U.S.A. & Canada surface; $46 Canada air mail; all other countries $52 per year. This fee includes $36/42/48 for *Forth Dimensions*. No sales tax, handling fee, or discount on membership.
When you join, your first issue will arrive in four to six weeks; subsequent issues will be mailed to you every other month as they are published—six issues in all. Your membership entitles you to a 10% discount on publications and functins of FIG. Dues are not deductible as a charitable contribution for U.S. federal income tax purposes, but may be deductible as a business expense.

MAIL ORDERS:
Forth Interest Group
P.O. Box 2154
Oakland, CA 94621
PHONE ORDERS:
510-89-FORTH Credit card orders, customer service. Hours: Mon–Fri, 9–5 p.m.

PAYMENT MUST ACCOMPANY ALL ORDERS

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

POSTAGE:
All orders calculate postage as number of #s times selected postage rate. Special handling available on request.

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

**** CALIFORNIA SALES TAX BY COUNTY:**
7.5%: Sonoma; **7.75%**: Fresno, Imperial, Inyo, Madera, Monterey, Orange, Riverside, Sacramento, San Benito, Santa Barbara, San Bernardino, San Diego, and San Joaquin; **8.25%**: Alameda, Contra Costa, Los Angeles San Mateo, Santa Clara, and Santa Cruz; **8.5%**: San Francisco; **7.25%**: other counties.

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

FIG Debriefing:

Embedded Systems Conference

John Rible
Santa Cruz, California

FIG was offered a booth at this year's Embedded Systems Conference "as an experiment" by the promoter, Miller-Freeman Publications. I took on the challenge to make it successful and succeeded, learning a lot in the process. We've been invited back next year, when it will be even better! What follows is an account of the process, in the hopes that other chapters will decide to do the same at conferences in their area.

The goal of our participation was to provide information about Forth to people outside the current user community. To achieve that at this conference meant showing Forth in embedded applications. Since F-PC and eForth don't address this area directly, I wanted to include vendors as much as possible, to show non-Forth users what was possible. To increase the excitement, I wanted a raffle and demonstrations. And I needed people to staff the booth. We also wanted to let chapter members know that they could get into the exhibits (and the parties!) free. No selling is allowed at this show, so there would be no piles of disks and books to worry about.

I wanted to include vendors and show non-Forth users what was possible.

So I called all the vendors I could think of who did cross-compilers or boards, about a dozen in all. I was overwhelmed by the enthusiasm! All of them wanted to participate, and in some cases were surprised that I didn't want to charge them. Three vendors (AM Research, New Micros, and Vesta) agreed to donate a board in return for the names of the entrants. Three vendors (AM Research, Forth Inc., and Mosaic Industries) agreed to demonstrate their systems, one each day. I arranged to have the literature sent to me the week before the show, guessing that 200–300 copies of brochures would be about right, ten per cent of the expected number of attendees. Since table space was limited, they were restricted to just one or two items each.

The people at Miller-Freeman were wonderfully supportive, and sent us, on very short notice, the "free pass" mailers to go out in our newsletter. At our chapter meeting the month before the conference, I signed up people to staff the booth. Then I called each of them the week before to confirm it. Most everybody showed up, some even for *much* more than they'd agreed to: it was fun! I got the Forth Interest Group office (they're local for us) to donate a couple of books for the raffle, make copies of FIG membership applications, and help out at the booth some, too. I made up business-card sized name-address-phone cards for people without business cards who wanted to enter the raffle, along with a sign for the jar.

I went the whole first day and half of the last day, to coordinate setup and cleanup. The hall had strict labor union requirements, so all the brochures were carried in and out by hand (or else *they* do it at $45/hour, one hour minimum) and the booth was set up without tools (same "or else"). We used wire racks to display vendor literature at one end; had FIG info, the *EE Times* "Forth in Space" article, "A Brief Intro to Forth" by Phil Koopman, Jr., and the raffle jar in the middle; and the vendor demo at the other end. We usually had two chapter members in the booth, with one or two vendor folks as well. The aisles were generally full, with people stopping almost continuously. It was very relaxed, though: people took breaks as desired and the staffing schedule was revised as each day went on.

After the show, I drew the winners from the raffle jar, packaged up the systems and sent them off. UPS was cheapest in the United States, and the U.S. Postal Service for England! The winners were:

Monday's FIG book:	Ian David, London, U.K.
Tuesday's FIG book:	Gary W. Dow, San Jose, CA
Wednesday's FIG book:	Doyle Kisler, San Jose, CA
Vesta system:	Ron Palmieri, Daly City, CA
NMI system:	Lennart Suurik, Sunnyvale, CA
AM Research system:	Richard Tobias, San Jose, CA

Al Mitchell of AM Research volunteered to enter the raffle names on disk. I contacted a couple of vendors to arrange the return of a lot of their expensive brochures. When the disk with the names arrived, I made copies and sent them to FIG and the donating vendors. I sent thank you letters to all the vendors as well.

Next time, I won't be so optimistic about how many copies to have: although there were 150 cards in the raffle jar, only 50–100 copies each of the various brochures were taken! I'll also collate the vendor brochures and slip them into a 17x11 folded sheet with FIG, local chapter, and the Intro to Forth information on it, so there's just one pile of info and more room for books and demonstrations. I hope that, by starting earlier, we'll be able to get the "free pass" out with *Forth Dimensions*. There won't be an X3J14 meeting to interfere with getting thank-you's out to the vendors. And I won't mistakenly have my home address embossed on the exhibitor badges!

See you there.

Placing Characters on the Screen

C.H. Ting
San Mateo, California

[The last tutorial demonstrated how to define new commands and how to use the string-printing function `."` *to generate block letters on the display—Ed.]*

In this lesson, we will try to write block characters anywhere on the screen. The screen displays characters in 25x80 format, that is, 25 lines with 80 characters per line. The following instructions allows us to position the cursor before writing characters:

`dark`	Clear screen and put cursor at top-left corner.
`at`	Move the cursor to specified screen location,
`40 12 at`	e.g., put the cursor at the center of the screen.

The following instruction puts a block-letter F at the center of screen:

```
: newBar         ." *****" ;
: newPost        ." *    " ;
: new-F
        dark
        38 10 at newBar
        38 11 at newPost
        38 12 at newBar
        38 13 at newPost
        38 14 at newPost
        38 15 at newPost
        cr
        ;
```

But it is very awkward to place a character by specifying the location of each of its separate elements. A more general way to place characters is to use variables to store the location, so that information isn't mixed in with the instructions that generate the character itself.

```
variable x
variable y
: newLine
  x @ y @ at  \ move cursor to x,y location
  1 y +!      \ increment y for next line
  ;
```

```
: F       newLine newBar
          newLine newPost
          newLine newBar
          newLine newPost
          newLine newPost
          newLine newPost
          ;
```

Now to place F on the screen, we first specify its location:
`30 x !   10 y !   F`

We have just used several more Forth instructions:

`variable <name>`	Define a variable where numbers can be stored and retrieved.
`@ ( var -- data )`	Fetch the number stored in a variable.
`! ( data var -- )`	Store a number into a variable.
`+! ( data var -- )`	Add a number to the value stored in a variable.

```
: F-demo
        dark
        0 x  ! 0 y ! F
        70 x ! 10 y ! F
        10 x ! 18 y ! F
        40 x ! 15 y ! F
        ;
```

Exercise One. Define a new instruction to clear the screen and put the message *FORTH* at the center of the screen in block characters.

```
: bar            ." *****" ;
: post           ." *    " ;
: triad1         ."  *** " ;
: sides          ." *   *" ;
: tetra          ." **** " ;
: duo1           ." **   " ;
: duo2           ." * *  " ;
: duo3           ." *  * " ;
: center         ."   *  " ;
: O      newLine triad1
         newLine sides
         newLine sides
         newLine sides
         newLine sides
         newLine triad1
         ;
: R      newLine tetra
         newLine sides
         newLine tetra
         newLine duo2
         newLine duo3
         newLine sides
         ;
: T      newLine bar
         newLine center
         newLine center
         newLine center
         newLine center
         newLine center
         ;
: H      newLine sides
         newLine sides
         newLine bar
         newLine sides
         newLine sides
         newLine sides
         ;
: FORTH dark
         25 x ! 10 y ! F
         32 x ! 10 y ! O
         39 x ! 10 y ! R
         46 x ! 10 y ! T
         53 x ! 10 y ! H
         ;

: demo   FORTH 20 8 62 17 box 0 21 at ;
```

Exercise Two. Design a message yourself and display it at the center of the screen.

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

New Contest Announcement

Call for Papers!

Forth Dimensions is sponsoring a contest for articles about

"Forth Development Environments"

Write about:
libraries, source management, user interfaces, platform/machine/kernel independence, other topics suggested for this year's FORML conference, or issues not specified here which, nevertheless, relate to the general theme.

Entries will be refereed. Papers to be presented at FORML are eligible, but must be completed and received at our office by the contest deadline.

Mail a complete hard copy and a diskette (Macintosh 800K or PC preferred) to the:

Forth Interest Group
P.O. Box 2154 • Oakland, California 94621

Cash awards to authors:

1st place:	***$500***
2nd place:	***$250***
3rd place:	***$100***

Deadline for contest entries is August 1, 1993.

Don't Expire!

Good things are coming, and your continued participation as a reader of *Forth Dimensions* is important to us. Please take a moment to renew or to extend your subscription. Mail or fax the mail-order form in this issue, or call the Forth Interest Group at 510-893-6784. Your comments are welcome.

Advertisers Index

The Computer Journal	*35*
FORML Conference	*44*
Forth Interest Group	*25, centerfold*
Harvard Softworks	*42*
Laboratory Microsystems	*35*
Miller Microcomputer Services	*12*
Silicon Composers	*2*

Principles of Metacompilation

B.J. Rodriguez
Hamilton, Ontario, Canada

J. Compiler Directives (`IMMEDIATE` words)

Most of Forth's control structures are implemented as compiler directives: `IF ... ELSE ... THEN`, `BEGIN ... UNTIL`, `BEGIN ... WHILE ... REPEAT`, `DO ... LOOP`. These words are executed, rather than compiled, at compile time, and are known in Forth as `IMMEDIATE` words.

Forth also allows the programmer to create new compiler directives by defining words with the `IMMEDIATE` attribute.

1. Use

The Forth compiler directives are used in the same manner when metacompiling as when compiling "normally". For example:

```
: name
  word  word  IF  word  word  THEN  ;
```

However, the definition of a compiler directive—an `IMMEDIATE` word—is somewhat different in the metacompiler. This is because two sets of actions need to be defined. First, the word's action when executed in the Target system, as part of the metacompiled application. Second, the action the metacompiler must take when it encounters the word.

Consider the `IF ... THEN` example. Suppose a new Forth kernel is being metacompiled. The result of the metacompiler is a dictionary of words, including a complete Forth compiler, that will run on the Target system. Later, the programmer using the Target system will write programs with `IF ... THEN`. So, a "Target" action for `IF` and `THEN` must be part of the Forth kernel.

But the Forth kernel itself contains many `IF ... THEN` constructs! These must be recognized by the metacompiler while the kernel is being compiled, and the appropriate branches and branch offsets for the Target machine must be compiled into the Target image. So, a "Host" effect on the Target image must also be defined for `IF` and `THEN`.

The same holds true for any `IMMEDIATE` word in the Target application.

a) Defining the Target action

This is straightforward. The Target's action is defined just as any other Forth word to be executed in the Target, i.e., as a colon or `CODE` definition. The only difference is that the "precedence bit" in the name must be set, to indicate that the word is `IMMEDIATE`.

The metacompiler's `IMMEDIATE` (in the `HOST` vocabulary) will set the precedence bit of the last word defined in the Target image.

So, the Target action is written:

```
: name
  word word ... word ;   IMMEDIATE
```

b) Defining the Host action

The Host action must be defined, in words known to the Host, describing what operations are to be performed on the Target image.

This is specified after the Target word is defined, using the word `ACTS:` to specify the Host action, and the word `IMPERATIVE` to indicate that it is a compiler directive.

```
: name
   word word ... word ;   IMMEDIATE
HOST ACTS:
host-word host-word ... ; IMPERATIVE
```

Generally, the "host-words" will be words from the metacompiler lexicon, which act on the Target image.

c) Defining a Target action only

It is possible to envision a case where a Target version of a compiler directive must be metacompiled, but the Host action is not needed.

For example, suppose a new Forth kernel is to be created which includes the unsigned loop word `/LOOP`. The eventual user of the Target Forth will want to use this word. But this word is used nowhere within the Forth kernel, so a Host action for `/LOOP` is not needed—the metacompiler will never be called upon to use it.

In this case, the `HOST ACTS: ... IMPERATIVE` clause may be omitted.

d) Defining a Host action only
It is also possible to envision a case where a Host action, but not a Target action, is required for a compiler directive.

For example, consider an embedded, "sealed" application, such as a microwave oven written in Forth and burned into PROM. Obviously, the end application will have no terminal, no programmer, and no means of extending the program. In such a case, the sizable part of the Forth kernel which implements the compiler can be omitted. And, with no means of compiling, there is no need for compiler directives.

What is needed is merely a word which executes in the Host. Such a word is defined with the "native" : (colon).

```
HOST
: name   host-word host-word ... ;
```

(The word `IMPERATIVE` is not required.) This word will be defined in the "mirror" vocabulary which has last been selected for `DEFINITIONS`, amidst all of the mirror words. So, although this word will reside in the Host's memory, it will appear in the Target's search order—exactly the desired effect!

2. Implementation

Building a compiler directive which will be used by the Target is straightforward. An ordinary colon or `CODE` definition is compiled in the Target image. Then, a special version of `IMMEDIATE` is executed by the Host whose function is to set the precedence bit in the Target image, in the last defined Target word. This is, of course, found from the `LATEST` which is maintained for the Target.

Building the compiler directive which will be used by the metacompiler is somewhat more involved. Bearing in mind the First Rule of Metacompiler Design, this discussion will focus on the concrete example of compiling the Forth phrase

```
IF   FOO   BAR   THEN
```

Figure Eight shows the data which must be compiled into the Target image by this phrase. `IF` must compile the Target's `0BRANCH`, a `CODE` word, and leave space for an offset. `FOO` and `BAR`, being ordinary Forth words, compile normally. `THEN` resolves the branch by patching the correct offset after the `0BRANCH`.

The Forth code to accomplish this in a resident compiling environment is:

```
: IF      COMPILE 0BRANCH   HERE   0 , ;
          IMMEDIATE
: THEN    HERE   OVER -   SWAP ! ;   IMMEDIATE
```

First, the explicit instructions to the Host, to have this effect in the Target image, must be defined.

`IF` in the Host must compile the Target's `0BRANCH`. Assuming, for the moment, the existence of a word such as `TCOMPILE`:
`TCOMPILE 0BRANCH`

The target's Dictionary Pointer must be stacked:
`HOST HERE` (recall that this is the Target DP)

Then the empty cell must be left for the offset:
`0 T,`

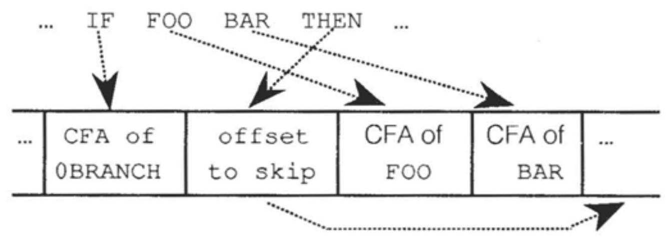
The branch will be resolved when `THEN` is encountered. The Target address of the offset cell is still on the stack. The current Dictionary Pointer is obtained:
`HOST HERE`

Then the offset is calculated in the Host...
`OVER -`

...and the result is stored in the Target image, at the address of the offset cell:
`SWAP T!`

Figure Eight. Compiling IF ... THEN (compiler directives).

In "normal" Forth...



Usually done by:

```
: IF ( -- a )   COMPILE 0BRANCH
  HERE 0 , ; IMMEDIATE

: THEN  ( a )   HERE OVER -
  SWAP ! ; IMMEDIATE
```

The Forth word `IMMEDIATE` means, when in compiling state, to execute this word instead of compiling its CFA.

To compile for the target...
...we would like the compile-time action to be like:

```
TCOMPILE 0BRANCH   T-HERE   0 T,
```

The problem of writing a TCOMPILE was glossed over in this discussion. Its function is to compile the following word—which is a Target word—into the Target image. But this is exactly the action of a mirror word. So the metacompiler need only ensure that the mirror word 0BRANCH is executed, and not the Host's "native" 0BRANCH. This is done:

```
TARGET 0BRANCH
```

When does the Host perform these actions? When it attempts to "compile" the words IF and THEN into the Target—i.e., when the metacompiler parses the words IF and THEN and attempts to execute their mirror words. (Recall that mirror words, when executed, compile their Target equivalents.) All that is necessary to change these words from "words which are compiled" into compiler directives, is a change in the run-time action of their mirror words in the Host.

Forth provides a mechanism for changing the run-time action of a word: DOES>. This implementation uses the Forth-79 DOES>, which acts by changing the CFA of the word in question. The new CFA points to a short machine code subroutine (DODOES>), which re-enters the Forth interpreter for the high-level code which follows.

So the Host word ACTS: should immediately execute DOES> to change the CFA of the latest word—a mirror word—and compile the DODOES> machine code. It then enters the compiling state in the Host to compile the words which describe the Host's action for that mirror word.

The result, for the word IF is shown in Figure Nine.

The advantages of metacompiling by executing mirror words should now be apparent!

This is not exactly how the Image Compiler handles compiler directives. The problem of Defining Words (next section) will introduce some new twists into the action of mirror words, and this will be reflected in the handling of compiler directives.

3. Alternatives

a) The absence of executable mirror words

As noted, some metacompilers do not execute their mirror words, or don't maintain a mirror vocabulary at all. In these metacompilers, the actions of all the Target compiler directives must be explicitly coded as part of the metacompiler (usually in the TARGET vocabulary, or its equivalent).

Some of these metacompilers have difficulty adding new compiler directives, once the metacompiler is completely loaded.

K. Defining Words

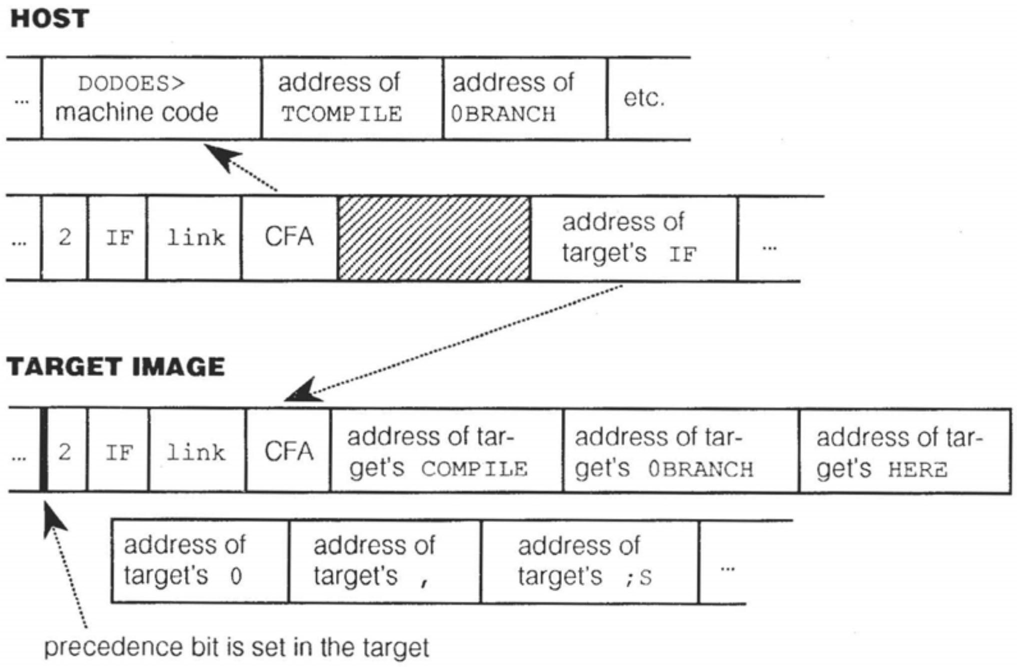
One of Forth's most powerful features is the ability to create new classes of words. A new class is described by a Forth "defining word," so named because it will be used to create the new Forth words which are members of the class.

Defining words are among the most powerful tools of the skillful Forth programmer, so it is a pity that many metacompilers provide little or no means of including them in a metacompiled application. Not so the Image Compiler.

1. Use

A brief refresher on defining words is in order here. Defining words use the <BUILDS ... DOES> construct in fig-Forth; CREATE ... DOES> in Forth-79 and Forth-83.

Figure Nine. Compiling IF ... THEN.



```
TARGET : IF COMPILE 0BRANCH
         HERE 0 , ; IMMEDIATE

HOST ACTS: TCOMPILE 0BRANCH
         HERE 0 T, ;
```

We need an IF to run in the target, and one for the host.

Figure Ten. Defining words.

`: CONSTANT`
`  <BUILDS ,   DOES> @ ;`
	Definition of `CONSTANT` for target machine to use.

`64 CONSTANT C/L`	1. Host uses `CONSTANT` to define a word in the target.
`C/L 16 * CONSTANT B/BUF`	2. Host uses a defined word (`C/L`) interpretively.
`: .LINE  ...  C/L TYPE ;`	3. Host compiles a defined word into a definition in the target.
`: FIELD  DUP + CONSTANT ;`	4. The word `CONSTANT` is compiled into a target definition, for later execution by the target.

In general, we need to code:
1. a `<BUILDS` action for the host to use
2. a `DOES>` action for the host to use
3. a `DOES>` action for the target to use
4. a `<BUILDS` action for the target to use

Three "sequences" are involved in their creation and use [7]:

Sequence One is when the defining word is itself built. This happens once.

Sequence Two is when the defining word is executed. This causes a "defined word"—a member of the class—to be built. This may happen many times, each time adding a word to the dictionary. The action to be taken during Sequence Two is specified by the `<BUILDS` clause.

Sequence Three is when a defined word is executed. The action which is taken during Sequence Three—the common action of the new class—is specified by the `DOES>` clause.

An example of a defining word is `CONSTANT` (Figure Ten). `CONSTANT` is itself a colon definition (Sequence One). Each time `CONSTANT` is executed (Sequence Two) it defines a new word, a named constant. The action of a named constant (Sequence Three) is to put its integer value on the stack.

Observe, in Figure Ten, the many ways a defining word and its "children" may be used while metacompiling a Forth kernel:

a) The Host may perform the defining action, e.g., to create a `CONSTANT` which is in the kernel.
b) The Host may need to execute a defined word, e.g., to get the value of one of the Target kernel's `CONSTANT`s.
c) The Host may need to compile a defined word into a Target colon definition.
d) The Host may need to compile the defining word itself into another colon definition. (This is not uncommon; many classes are built using `CONSTANT` and specifying a different `DOES>` clause.)

Of course, in this example of a new Forth kernel, the eventual user of the Target system will need all the components of the defining word.

To cover all contingencies, the metacompiler must be able to define:

a) a `<BUILDS` action for the Host to use;
b) a `DOES>` action for the Host to use;
c) a `<BUILDS` action for the Target to use; and
d) a `DOES>` action for the Target to use.

The Image Compiler syntax for all of these operations is:

```
: name    <BUILDS   word word word   (c)
      DOES>  word word word  ;       (d)
  HOST ACTS:    word word word       (a)
  HOST DOES>    word word word  ;    (b)
```

Any of these clauses may be omitted. Particular instances where this would be useful are:

a) The Host `<BUILDS` is not needed if this defining word will not be used during the metacompilation. For example, `2CONSTANT` could be included in the kernel; but it is not used during the compilation of the kernel.
b) The Host `DOES>` action is not needed if no defined word will be used interpretively during the metacompilation. For example, `C/L` is defined in the fig-Forth kernel, but its only use is when it is compiled into Target colon definitions.
c) The Target `<BUILDS` action is frequently omitted in embedded applications. No compilation takes place in the Target system; all defined words are created during the metacompilation.
d) It is possible, though rather pointless, to omit the Target `DOES>` action. (Why build a word in the Target image that only the Host can use?)

The defined word action in the Target may also be written in machine code, as:

```
: name    <BUILDS   word word word   (c)
      ;CODE   assembly code          (d)
  HOST ACTS:    word word word       (a)
  HOST DOES>    word word word  ;    (b)
```

There is no provision to specify the Host's `DOES>` action in machine code; blinding speed is not usually required during metacompilation. Should it be necessary, a separate `CODE` definition can be made in the Host, and used in the `DOES>` clause.

2. Implementation

Note, from the examples given above and in Figure Ten, that the Host must be able to compile or to execute both defining and defined words.

Compiling a word into the Target image is done by executing its mirror word. Both the defining and the defined words will have corresponding mirror words. But giving a Target word an executable behavior—as in the case of compiler directives—involves changing the run-time action of the mirror word.

The solution is to give the mirror words both a *compiling* action and an *executing* action, both of which can be altered.

This, of course, requires some means of distinguishing between the *metacompiling* and *meta-executing* states. Thus the variable `STATE` is resurrected.

Figure 11 illustrates the form now taken by the mirror word. (The reason for reserving a cell should now be clear.) The compiling action is specified by the code address of the word; this points to machine code in the Host, and is changed with `DOES>` (see the previous section). The executing action is specified by a common Forth execution vector, which points to a Forth word in the Host. This difference is because the former is invoked by the Forth interpreter, while the latter is invoked by the phrase `@ EXECUTE` .

The final definition of `CONSTANT` is given in Figure 11; the resulting code in Host and Target—including one instance of a defined word—is shown in Figure 12. A step-by-step analysis of the process follows.

Sequence 1: When the code is metacompiled,

`: CONSTANT`
Builds the colon definition header (`8 CONSTANT` link cfa) in the Target, and the mirror word named `CONSTANT` in the Host. The compile action of the mirror word is the default; the execute action is an error word.

`<BUILDS ,`
Are compiled into the Target image.

`DOES>`
Compiles the Target `(;CODE)` and the `DODOES>` machine code into the Target image.

`@ ;`
Are compiled into the Target image.

`HOST ACTS:`
Changes the execute vector of the mirror word to point to the following code; builds a headerless colon definition in the Host by compiling the address of the `(:)` machine code.

`CREATE T,`
Are compiled into the Host.

`DOES>`
Compiles `(DOES>)` (described below) and `;S` to end the colon definition in the Host. Then begins a new headerless definition, which will be the execution action (in the Host) of the "children."

`T@ ;`
Are compiled into the Host (`;` compiles `;S`).

Sequence 2: When `64 CONSTANT C/L` is executed interpretively, control is transferred to the first headerless definition described above.

`CREATE`
The metacompiler's `CREATE`—it builds a header in the Target image, and a mirror word of the same name (`C/L`) is built in the Host.

`T,`
Compiles the constant value (64) into the Target image.

`(DOES>)`
Which was compiled "invisibly" into the Host by `DOES>`, changes the execute vector of the new mirror word (`C/L`), to point to the second headerless definition described above. It also changes the code address of the new word in the Target image, to point to the `DODOES>` portion of the Target's `CONSTANT`.

Sequence 3: When `C/L` is executed interpretively, control transfers to the second headerless definition. (It is entered with the address of the mirror word on the stack.)

`T@`
Fetches the constant value (64) from the Target image, and puts it on the Host stack.

Finally, observe that the Target image has exactly the form required by the `<BUILDS ... DOES>` construct (with the Forth-79 enhancement). Both `CONSTANT` and `C/L` can be used in the new Forth kernel.

3. Alternatives

This implementation, having distinct and separate compiling and executing actions for the mirror words, is unique to the Image Compiler.

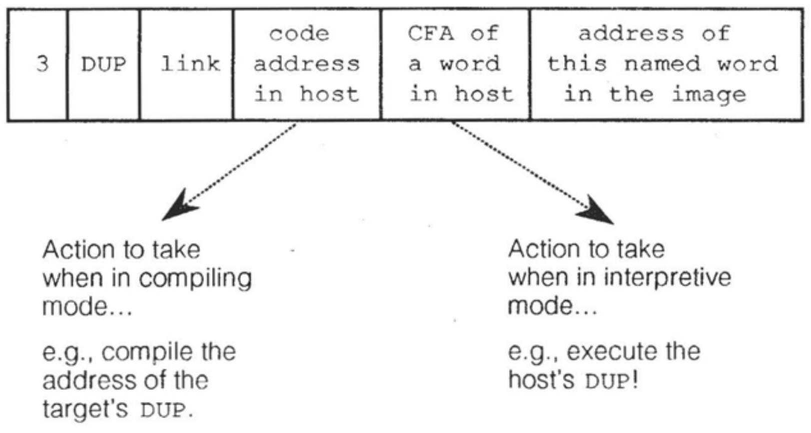
L. Compiler Directives Revisited

1. Implementation

Figure 11 also alludes to a fringe benefit of this approach. Since each mirror word has a separate executing vector, any word in the Target image can be given an interpretive action in the Host.

For example, the mirror word `DUP` normally compiles the address of the corresponding `CODE` word in the Target image. It may be desirable to use `DUP` interpretively—

Figure Eleven. Improved mirror word in the Host.


3
DUP
link
code address in host
CFA of a word in host
address of this named word in the image
Action to take when in compiling mode...
e.g., compile the address of the target's DUP.
Action to take when in interpretive mode...
e.g., execute the host's DUP!

The host must keep a metacompiler STATE!

What we would like to write:

```
TARGET : CONSTANT  <BUILDS ,      [4]
                     DOES> @ ;   [3]
              HOST <BUILDS T,     [1]
                     DOES> T@ ;  [2]
```

affecting the Host stack, of course—during metacompilation. By setting the executing vector of the mirror word to point to the Host `DUP`, this is achieved.

There are so many cases like this, where a mirror word's execution is vectored to a single Host word, that a special version of `ACTS:` is defined:

`HOST ACT word`
Changes the execution behavior of the latest mirror word, to the single Forth word in the Host.

M. Forward Referencing

Forth as a language provides no formal support for forward references. All words must be defined before they are used. So it is paradoxical that the Forth kernel itself depends on the use of forward references. This problem must be considered in the metacompiler.

1. Use

Forward referencing is automatic in the Image Compiler. A word which is to be forward referenced may be used in a colon definition just like any other word:

```
: name   word  ...  fwd-word  ... word  ;
```

The Image Compiler will compile an empty cell in the place of `fwd-word`. Later, when `fwd-word` is defined, its address will automatically be placed in this definition.

The following rules and restrictions apply:

a) Forward references can only be made within colon definitions.
b) Compiler directives and other `IMMEDIATE` words may not be forward referenced.
c) The same word may be forward referenced any number of times.
d) The first definition having that name will be the definition used to resolve any forward references.
e) The usual logic of search order in Forth vocabularies does not apply to forward referencing. For safety, forward-referenced words should be uniquely named, and forward references should not cross vocabularies. Doing otherwise may lead to unpredictable results.
f) All forward-referenced words must eventually be defined!

While forward referencing is active, there is no such thing as an undefined word—words are either defined, or are expected to be defined later. When forward references are not being used, this will deprive the programmer of useful diagnostic information. So, forward referencing can be enabled and disabled with

`FORWARD ON` and `FORWARD OFF`

2. Implementation

The Image Compiler handles forward references by having the Host remember the location of all references to an unknown word. When that word is later defined, all the remembered locations can be patched with its address.

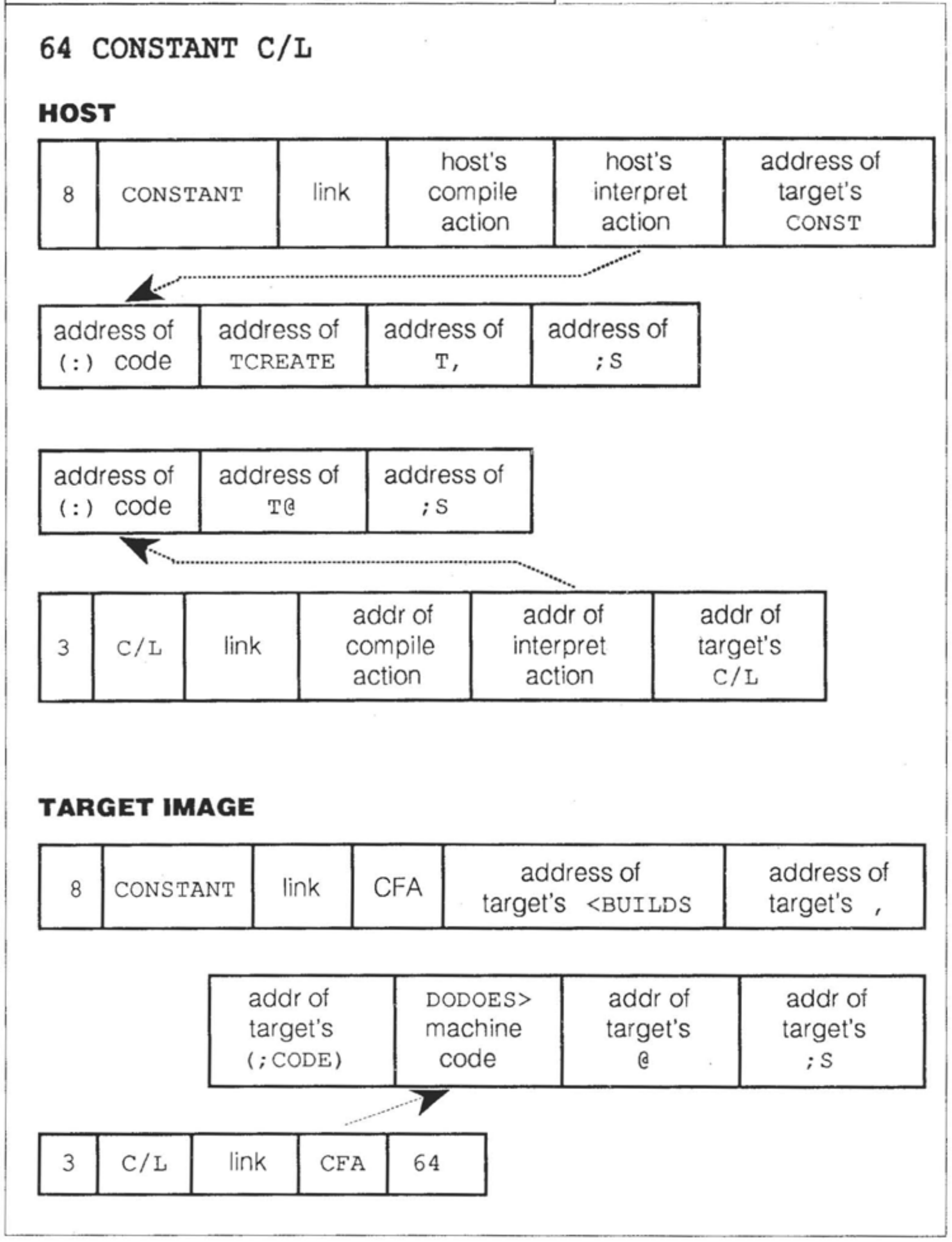
First reference. Figure 13 illustrates the Host's processing the first time a word is forward-referenced. When the word is encountered in the input stream, the Host will attempt to find and execute the mirror word of that name in the `TARGET` vocabulary (or sub-vocabulary). Failing this, the Host will attempt to convert it as a number. Failing this, the word is considered undefined, and the Host presumes this to be a forward reference.

An empty cell is compiled into the Target image, where the address of this unknown word belongs. Then the Host creates a dictionary entry, in the `CURRENT` mirror vocabulary, using the name of the unknown word. This is how the Host remembers the name of this word, so that it will be recognized when it is defined later. This dictionary entry—a forward-referencing mirror word—contains a pointer to the Target location to be patched.

Second and subsequent references. The next time that word is used, it will be found in the dictionary! What is found is not the normal, self-compiling mirror word, but the forward-referencing mirror word described above. This word is given a special compile-time action:

a) Reserve an empty cell in the target, to be patched later.
b) Store the address of the previous location-to-be-patched (as found from the mirror word) in this reserved cell.
c) Change the mirror word to point to this new reserved cell.

Figure Twelve. Defining words—the result.



the links.

Finally, a normal self-compiling mirror word is created for the new definition.

Note that any Forth word may be forward-referenced in this manner: colon definitions, CODE definitions, data structures, or defined words.

Later "backward" references. Thanks to the search order of Forth, all dictionary searches for this word will now stop at the new, normal, mirror word. To all appearances, the normal mirror word completely replaces the prior, forward-referencing mirror word.

Thus, when the word is next encountered in the metacompilation, the normal mirror word corresponding to its definition will be found and executed, compiling its Target address. No other forward references will be made for this word.

Which is exactly the desired result.

3. Issues

a) Disabling forward referencing

In most application programming, an undefined word is an error condition and should be reported as such. Forward referencing interferes with this.

Forward referencing is disabled by setting a flag. This flag must be tested in two places: to prevent a forward-reference mirror word from being constructed on the first occurrence; and within that mirror word, to prevent subsequent occurrences from being linked into the list. Forward referencing can be turned on and off many times within the metacompilation.

The effect of these actions is to build, in the Target image, a linked list of all cells to be patched with the address of this unknown word. The forward-referencing mirror word contains the pointer to the head of the list. Each different forward-referenced word (i.e., each unknown name) will have a separate linked list. See Figure 14.

Definition of the word. When the word is finally defined, TCREATE will find its name is already in use. Before reporting this as a duplicate name (a Forth re-definition), TCREATE checks to see if the prior use is a forward-referencing mirror word. (This test is performed by checking the execution vector, since forward-referencing mirror words have a specific and unique action.)

If there is a forward reference, the head of the linked list is fetched from the mirror word. The address of the new definition is patched into all the entries of the list, replacing

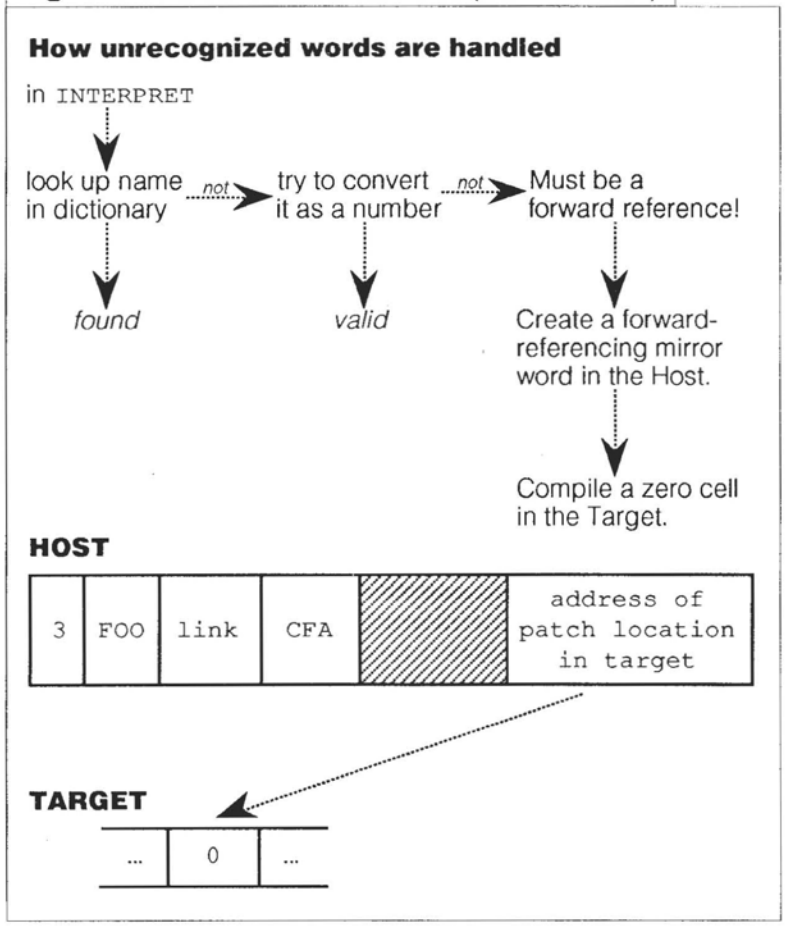
4. Alternatives

a) F83

The Laxen-Perry F83 metacompiler uses a forward-referencing scheme similar to that of the Image Compiler, except that it is manual instead of automatic.

Each word which will be forward referenced must be declared prior to its first appearance, by the phrase

Figure Thirteen. Forward references (first occurrence).


How unrecognized words are handled
in INTERPRET
look up name in dictionary
not
try to convert it as a number
not
Must be a forward reference!
found
valid
Create a forward-referencing mirror word in the Host.
Compile a zero cell in the Target.
HOST
3
FOO
link
CFA
address of patch location in target
TARGET
...
0
...

N. Getting the Image into the Target

The Image Compiler is now complete. It is capable of metacompiling programs ranging from a few hundred bytes of embedded application code, to a full Forth kernel, to a complex Forth application with multiple vocabularies and user-created defining words. What remains is the problem of getting this metacompiled code into the Target system!

1. Via hex file

Perhaps the most useful method in the IBM PC environment is to create an MS-DOS file with the Target's binary image in some generally recognized format.

The Intel hex format is widely accepted. Many operating systems can convert Intel hex files to executable files. Most EPROM programmers, and many resident monitor programs (e.g., the Zilog Super8 Monitor) will accept the Intel hex format.

The word HEXFILE (screen 43) will copy a range of Target memory to an MS-DOS file, in the Intel hex format:

```
address length HEXFILE name
```

The starting address is the origin of the metacompilation—set in the source file, before the first definition, by the phrase:

```
DEFER name
```

DEFER builds a word in the Host which maintains a linked list of forward references, in essentially the same manner as the Image Compiler. After the forward-referenced word is actually defined, the forward references must be explicitly resolved by the phrase

```
' defined-name RESOLVES reference-name
```

which has the unfortunate consequence of requiring the name used for forward references to be different than the name used in the eventual definition.

b) Metaforth
Metaforth and many similar metacompilers require all forward references to be explicitly patched by the programmer. This usually means knowing the location to be patched as a byte offset within a colon definition. If the definition is changed, the patch offset must be edited accordingly. Normally, all of the patches are performed, in a load screen, as the last step of metacompilation.

To reserve the empty cell which will be patched, a metacompiler directive (often called GAP) is used.

```
address HOST DP !
```

It is frequently convenient to define an EQUate to contain this address, so that it may be referenced by some easily remembered name (like ORIGIN) after the compilation is finished.

At the end of the compilation, the first unused address in the Target dictionary can be obtained with:

```
HOST HERE
```

and thus the length of the image can be found with:

```
HOST HERE ORIGIN -
```

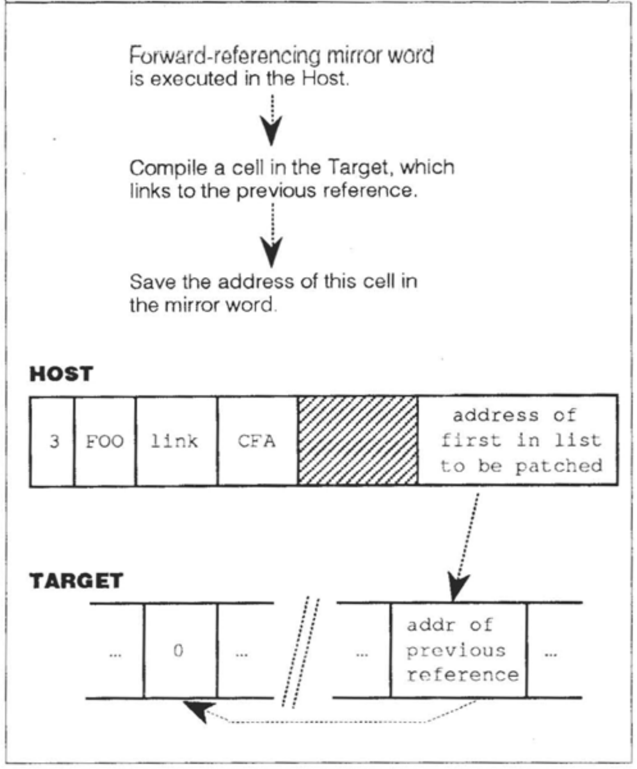
(It would be a simple exercise to write this as a Forth word; but so far, it's been advantageous to know these addresses before doing the download.)

2. Direct download to Target or EPROM

It would be possible to write a program, in Forth, which communicates directly with the Target system (via its resident monitor program) or an EPROM programmer.

The difficulty is that every target system and every EPROM programmer has a slightly different protocol for communication over a serial link. So a different download program is necessary for each different piece of target

Figure Fourteen. Forward references (subsequent).


Forward-referencing mirror word is executed in the Host.
Compile a cell in the Target, which links to the previous reference.
Save the address of this cell in the mirror word.
HOST
3
FOO
link
CFA
address of first in list to be patched
TARGET
...
0
...
...
addr of previous reference
...

Metacompilation code, conclusion.

```
screen # 87
   ( Super8 [compile] ( )   HEX               (  8  6 88 bjr 12:39 )
TARGET : [COMPILE]   -FIND 0= 0 ?ERROR DROP CFA , ;  IMMEDIATE
HOST ACTS:   DROP -FIND 0= 0 ?ERROR DROP 2+ @ T, ;  IMPERATIVE

TARGET : (   29 WORD ;  IMMEDIATE
HOST ACTS:   DROP 29 WORD ; IMPERATIVE
;S


screen # 88
   ( Super8 do loop if else then)             (  8  6 88 bjr 12:29 )
TARGET : BACK   HERE - , ;
TARGET : DO      COMPILE (DO) HERE 3 ; IMMEDIATE
  HOST ACTS: DROP  TARGET (DO) HOST HERE ; IMPERATIVE
TARGET : LOOP    3 ?PAIRS COMPILE (LOOP)  BACK ; IMMEDIATE
  HOST ACTS: DROP  TARGET (LOOP)  HOST HERE - T, ; IMPERATIVE
TARGET : +LOOP   3 ?PAIRS COMPILE (+LOOP) BACK ; IMMEDIATE
  HOST ACTS: DROP  TARGET (+LOOP)  HOST HERE - T, ; IMPERATIVE
TARGET : IF      COMPILE 0BRANCH HERE 0 , 2 ; IMMEDIATE
  HOST ACTS: DROP  TARGET 0BRANCH  HOST HERE  0 T, ; IMPERATIVE
TARGET : THEN    ?COMP 2 ?PAIRS HERE OVER - SWAP ! ; IMMEDIATE
  HOST ACTS: DROP  HERE OVER - SWAP T! ; IMPERATIVE
TARGET : ELSE    2 ?PAIRS COMPILE BRANCH HERE 0 ,
   SWAP 2 [COMPILE] THEN 2 ; IMMEDIATE
  HOST ACTS: DROP  TARGET BRANCH  HOST HERE 0 T,  SWAP
   HERE OVER - SWAP T! ; IMPERATIVE                           ;S


screen # 89
   ( Super8 begin - repeat)                   (  8  6 88 bjr 12:28 )
TARGET : BEGIN    ?COMP HERE 1 ; IMMEDIATE
  HOST ACTS: DROP  HERE ; IMPERATIVE
TARGET : UNTIL    1 ?PAIRS COMPILE 0BRANCH BACK ; IMMEDIATE
  HOST ACTS: DROP  TARGET 0BRANCH  HOST HERE - T, ; IMPERATIVE
TARGET : AGAIN    1 ?PAIRS COMPILE BRANCH  BACK ; IMMEDIATE
  HOST ACTS: DROP  TARGET BRANCH  HOST HERE - T, ; IMPERATIVE
TARGET : WHILE    [COMPILE] IF 2+ ; IMMEDIATE
  HOST ACTS: DROP  TARGET 0BRANCH  HOST HERE  0 T, ; IMPERATIVE
TARGET : REPEAT   >R >R [COMPILE] AGAIN R> R> 2-
   [COMPILE] THEN ; IMMEDIATE
  HOST ACTS: DROP  SWAP  TARGET BRANCH  HOST HERE - T,
    HERE OVER - SWAP T! ; IMPERATIVE

;S : END   [COMPILE]  UNTIL ; IMMEDIATE
   : ENDIF   [COMPILE] THEN ; IMMEDIATE
```

hardware. (When hex files are used, these differences are handled by the manufacturer-supplied host software.)

Still, for a frequently used target device, the time savings in being able to download directly from Host memory would make a Forth download program useful.

3. Metacompiling to Target memory

Carrying this logic one step further, if the Target system has a resident monitor program which allows memory to be examined and altered over a serial link, and if this system is connected to the Host at the time of the metacompilation, it would be possible to metacompile directly into the Target's memory!

This obviously would necessitate rewriting the Target memory words (`T@`, `T!`, etc.) to transmit commands and parse responses over the serial link. A slight speed penalty is involved, but since memory transfers are not the critical element in compile time, the degradation may not be noticed.

This leads to possibilities of fully interactive metacompilation, where words can be compiled one at a time in the Target, tested individually, forgotten, and redefined... making the metacompiler environment every bit as interactive as a normal Forth system!

O. References

METAFORTH is a trademark of John J. Cassady.
Target Compiler is a trademark of FORTH, Inc.

1. Rodriguez, B. J., "B.Y.O. Assembler," *The Computer Journal* #52 (Sept/Oct 1991).
2. Rodriguez, B. J., "B.Y.O. Assembler: A 6809 Forth Assembler," *The Computer Journal* #54 (Jan/Feb 1992).
3. Ewing, Martin S., *The CalTech Forth Manual*, a Technical Report of the Owens Valley Radio Observatory, California Institute of Technology, Pasadena, CA (2nd ed., June 1978).
4. Laxen, Henry, "Techniques Tutorial: Meta Compiling I," *Forth Dimensions* IV/6 (Mar-Apr 1983). Discussion of host and target memory spaces.
5. Laxen, Henry, "Techniques Tutorial: Meta Compiling II," *Forth Dimensions* V/2 (July-August 1983). Compilation of CODE and colon definitions.
6. Laxen, Henry, "Techniques Tutorial: Meta Compiling III," *Forth Dimensions* V/3 (Sept/Oct 1983). Forward references and compiler directives.
7. Derrick, Mitch and Baker, Linda, *FORTH Encyclopedia*, Mountain View Press, Mountain View, CA (1st ed., 1982).
8. Cassady, John J., *METAFORTH*, Mountain View Press, Mountain View, CA (1st ed., 1980).
9. Walker, Ray and Rather, Elizabeth, *polyFORTH II Reference Manual*, FORTH, Inc., Manhattan Beach, CA (4th ed., 1983).
10. Ragsdale, William F., "The 'ONLY' Concept for Vocabularies," *1982 FORML Conference Proceedings*, Forth Interest Group, San Carlos, CA (1982).
11. Rodriguez, B. J., "Interactive Embedded Metacompilation," *Proceedings of the 1990 Rochester Forth Conference*, Institute for Applied Forth Research, Rochester, NY (1990).

(Letters, continued from page 5.)

Fast 32-bit Integer Square Root

Dear Marlin,

This is a simple—even primitive—contribution, but it may be useful to some Forth users.

I have been writing an application in JForth (Delta Research) and, for space reasons, I did not want to use JForth's floating-point library. I derived integer trigonometrical functions from the article by Phil Koopman, Jr. (*FD* IX/4), but I also needed a 32-bit integer square root function. Delta Research supplies a Newtonian successive approximation square root utility which is credited to R.L. Davies (*FD* VII/4). Despite an added convergence test, the result is relatively slow, averaging over three milliseconds per root.

Back in the days when the Intel 8080 represented the state of the art, I derived a direct computation square root algorithm. This I have disinterred and converted to Forth. Though it always executes a loop 16 times, it uses no divisions. On my standard Amiga 500, it computes the root of a 32-bit number in under a millisecond. It should not prove too difficult to extend it to 64 bits.

Since it lies in an inner loop in my application, I plan to implement this algorithm in 68000 machine language. However, its incarnation in Forth may be of use to others as a program speed-up device. As shown here, it returns both the root and the remainder. The latter is the difference between the input number and the square of the root., It may be used to round up the root, or can be dropped, as the users wishes. A zero root and a negative remainder indicate that the input number was negative.

Kind regards,

Tom Napier
One Lower State Road
North Wales, Pennsylvania 19454

32-bit Integer Square Root

```
: SQRT ( 32-bit number -- root remainder )
\ Root square plus remainder equals input
\ In comments, P is the processed input,
\ Q is the result, and S is a power of two
\ which steps by 1/4 per loop

  DUP 0<        \ test for negative input
  IF DROP 0 -1 \ flag negative input
  ELSE 0             \ initial Q
    SWAP 1073741824   \ initial S, 2^31
    BEGIN >R    \ keep S on return stack
      DUP 2 PICK - R@ -      ( Q  P  P' -- )
      DUP 0<    \ is P' (= P-Q-S) negative?
      IF DROP            \ restore P
        SWAP 2/          \ Q := Q/2
      ELSE NIP           \ use P'
        SWAP 2/ R@ +  \ Q := Q/2 + S
      THEN SWAP  R> 2/ 2/    \ S := S/4
      DUP 0=                 \ is S zero yet?
    UNTIL DROP               \ dump S
  THEN  ;
```

Total control with *LMI FORTH*™

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

For Development:
Interactive Forth-83 Interpreter/Compilers for MS-DOS, OS/2, and the 80386
- 16-bit and 32-bit implementations
- Full screen editor and assembler
- Uses standard operating system files
- 500 page manual written in plain English
- Support for graphics,floating point, native code generation

For Applications: Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 8088, 68000, 6502, 8051, 8096, 1802, 6303, 6809, 68HC11,34010, V25, RTX-2000
- No license fee or royalty for compiled applications

LMI *Laboratory Microsystems Incorporated*
Post Office Box 10430, Marina del Rey, CA 90295
Phone Credit Card Orders to: (213) 306-7412
FAX: (213) 301-0761

Free! Trial Subscription

There are whole other worlds in micro computers than DOS and Windows. If embedded controllers, Forth, S100, CP/M or robotics mean anything to you, then you need to know about *The Computer Journal.*

Hardware projects with schematics, software articles with full source code in every issue. And you can try *The Computer Journal* without cost or risk! Call toll free today to start your trial subscription and pay only if you like it.

Rates: $18/year US; $24/year Foreign. You may cancel your subscription without cost if you don't feel The Computer Journal is for you. Published six times a year.

(800) 424-8825

TCJ The Computer Journal
The Spirit of the Individual Made This Industry
Socrates Press
PO Box 535
Lincoln, CA 95648

Exploring Forth Issues and Promoting Forth

Fast FORTHward

Mike Elola
San Jose, California

To gain acceptance, a programming language needs to satisfy many needs. Two of these needs are interrelated and lead to increased programmer productivity.

First, a programming language should offer native functions that impart as much utility as possible. The uniform delivery of this functionality is of keen importance so that source code portability is maximized and so that language dialects are minimized.

Language uniformity is partly a function of how well the language implementors breathe life into a language. Nonetheless, a clear and well-understood language specification offers the best hope of establishing uniformity amongst implementations. It is in the best interests of everyone that agreement is reached over the language and that the language specification is elaborated in one document. (The ANS Forth specification offers the best hope for Forth to make progress regarding this critical first step to programming language acceptance.)

By delivering uniform and widely useful functions, a programming language synchronizes application code to a single, consistent processing model. This is an important advantage that high-level languages have over assembly language—particularly for the porting efficiencies that are possible.

Code relocatability and intermodule routine calling are the thorny issues...

When application code is written in a high-level language, at least two layers of functionality exist. Low-level functionality is consolidated in the language kernel, which is not dependent on the application. Even this gross partitioning reaps a reuse benefit that cannot be obtained in assembly language. Most of the porting effort for such an application will be the creation of a language compiler for a new processor. However, the language port has great reuse potential, so that work is likely to be amortized across many projects. Moreover, most of the routines developed in a high-level language are readily reusable—regardless of the platform they were developed for originally. In these ways, a high-level language can fuel efficiencies that cannot be equaled by any assembly language.

A programming language takes a second big stride toward acceptance by providing a facility to subdivide programs into modules with distinct interfaces. If even gross partitioning of an application (into a kernel component and a programmer-supplied component) has high code-reuse significance, then a more granular partitioning can lead to greater opportunities for code reuse.

You should be able to create these modules in a uniform way. You should have a uniform way to engage code purchased from a library vendor. Furthermore, you should have a compiler that can determine when the module's interface is not being honored properly. These measures help you reuse code written by others. The efficiencies that can be gained through such combinations of measures is widely acknowledged. Look at the advertisements in other programming journals as well as the growing number of articles discussing the creation and use of libraries.

Languages designed with consideration for such needs will include a formal lexicon for declaring the interface (or usage rules) for newly developed routines or modules. Besides establishing the beginning and end of a module, you may also need a way to hide elements that are not part of the module's visible interface, such as instances of data structures.

As a library user, you do not necessarily need to see source code. For the sake of writing portable code, you should not have to be concerned with the implementation details of either the language or any libraries you might be using. Offering libraries in a precompiled form suits that purpose. Only the source code that defines the interface has to be offered to enable you to correctly use the module.

The source code that sets up the interface for a module also provides information the compiler can use to help ensure that the interface is used properly. Many languages can verify that you are using the correct names for the precompiled routines as well as using the correct number and type of parameters.

Enhanced support for modules (or code reuse) helps create the commerce that library vendors enjoy. The library vendor becomes preoccupied with porting the same module to various platforms and with assuring that a uniform interface is served to each application despite any platform differences.

Adding support for modules does not automatically dictate that we transform modules into libraries. Without adding all the trappings of a library, separate or redundant declaration of the module interface is still possible: (1) in the central location where the complete source code for the module lives, and (2) as many client locations where source code must be compiled that exercises the module interface.

The inability to precompile shared modules may not be a siginificant concern. However, the ability to independently compile modules does become a big concern when projects require more than one programmer. Code relocatability and intermodule routine calling are probably the thorny issues to deal with here, not module precompiling itself. I predict that dynamic libraries will be extremely popular; so when Forth's full-blooded support of modules finally arrives, it should include a means for late binding of a routine inside a module to its callers outside the module.

By making it easier to reuse code across many different projects, module support that stops short of library support still offers a valuable boost—even for a solitary programmer. At the very least, such support should involve a uniform lexicon for declaring the interfaces for routines.

Implementation hiding occurs whether functions are standardized as a native component of a language or if they are precompiled and made accessible through its declared interface (for which the compiler can check client code for compliance). Either way the programmer need not be concerned with how the function is actually coded, just how it is engaged to suit the syntax requirements of the language or to suit its explicitly declared interface.

Furthermore, the portability of vast amounts of application code is improved, due to the uniform functions of a standard language or the frequent use of functions that serve as a module interface. That uniformity can help commit us to more uniform ways of specifying certain functions, reducing the amount of gratuitous diversity exhibited by our collective application code. A directly related effect is the improved readability of all application code that makes use of popular modules. The contemplated increase in the readability of Forth applications can only come about in such a way—no substitute approach appears to exist that comes anywhere near having the desired effect.

This discussion has not paid adequate homage to Forth's virtues in areas related to program partitioning and code reuse. I have offered a viewpoint slanted towards the perceptions and understanding of the larger programming community. Perhaps another essay will be written to offer an opposing, or Forth-centric, point of view.

Nonetheless, mainstream programming languages and mainstream programming practices recognize the need for separate declaration of interfaces. C now has function prototypes to add to the long-standing practice of using header files for ease of reuse of data type declarations. Those reused data type declarations help synchronize the structure of any data that must remain visible to client routines—or synchronize the data structures for which pointer references must be passed, as parameters to client routines or as return values from client routines.

If only to counter the perception of Forth as an outdated language, we need to transform stack comments into a formal, standardized mechanism for interface declaration that can be scaled up to the module level.

ANS Forth Update

The X3J14 committee convened in October 1992 to respond to comments resulting from the two-month public review ending in August of that year. That review period was necessary due to substantive changes made after the first four-month public review period ending in February of 1992. Because a few substantive changes were made at the October meeting, there will be yet another two-month public review in early 1993.

A January meeting is taking place in Los Angeles, California to contemplate a reorganized and reformatted version of the current draft proposal.

Mitch Bradley reports fewer comments (15) received in the second public review cycle, and fewer substantive changes made in its aftermath. Accordingly, committee members are optimistic about the prospect of obtaining ANS approval for a Forth standard sometime in 1993.

Among the substantive changes that came about as a result of the second review period are the obsolescence of `TIB` and `#TIB` (they are not immediately eliminated, but are among the controlled extensions). The single word `SOURCE` is their replacement. `SOURCE` returns an address and a length. A new word, `SLITERAL`, was added to compile strings. Also, a newly added clarification requires that an interpretive mode be present before compliance with ANS Forth is satisfied.

Product Watch

SEPTEMBER 1992

Laboratory Microsystems announced WinForth, a Forth-83 implementation of Forth that takes advantage of the graphical user interface provided by Microsoft Windows (version 3.1) in protected mode. It supports all Windows API functions, such as callbacks, dialogs, menus, and icons. Traditional command-driven applications are also supported through words such as `KEY` and `EMIT` in tandem with a resizable console window. No royalty or licensing fee is required to distribute Windows applications that you create with WinForth. Prices range from $100 for the basic version (on-line hypertext documentation only), to $495 for a professional version that supports DLL creation and includes source code for supplied utilities. For users of the professional version, the complete source code— including C, MASM, and Forth code—is also available for an additional premium. Upgrade prices are offered to registered users of UR/Forth and PC/Forth. The prices of the upgrades range from nothing to $250.

FORTH, Inc. announced a new release of its 32-bit polyFORTH software development system. Besides running in protected mode with DOS 5.0, it works with XMS as well as VCPI servers. This release sports enhanced editors and utilities for source code maintenance. Also included in the $1,495 price is a multitasking real-time executive, documentation, complete source code, and the ability to link to subroutines written in C and other languages.

OCTOBER 1992

FORTH, Inc. announced a new release of its EXPRESS Event Management and Control System for process control and factory automation applications. Besides enhancements and new I/O drivers, a historical trend recording feature is added. New driver support is added for OPTO-22 Optomux and Modicon V984.

ExpressLite, a $195 demonstration version of EXPRESS now supports limited (digital and analog) inputs as well as up to 256 simulated inputs. Use a PC to control simple experiments or devices through a graphical display that you can tailor for a simple apparatus controllable by one analog output and two analog inputs, or eight digital outputs and inputs. EXPRESS retails for $6,875, while ExpressLite retails for $195. Both systems require 80386/486 computers with 4 Mb of RAM and VGA graphics display.

Companies Mentioned

Forth Inc.
111 N. Sepulveda Blvd.
Manhattan Beach,
California 90266-6847
Fax: 213-372-8493
Phone: 800-55-FORTH

Laboratory Microsystems Inc.
12555 W. Jefferson Blvd.,
Suite 202
Los Angeles, California 90066
Fax: 310-301-0761
BBS: 310-306-3530
Phone: 310-306-7412

("Back Burner," from page 43.)

Back to the Future

Sometimes the path of advance is a seeming regression. For many applications, the proper solution to the problems of real-time control and instrumentation lies not in the application of faster computers addressing more memory but, rather, in simplification of directly interfaced hardware and software. In an interrupt-driven environment, even a slow eight-bit processor may prove more effective than a fast 386 PC-compatible.

When designing a computer system for control and instrumentation, maintainability demands that the critical portion of the system consist of components designed and standardized for industrial service. Such components may be in the form of standalone boards or they may be bus oriented. The most widely used industrial bus is the STD bus. With support from a large number of domestic manufacturers, an installed base second only to the IBM-PC, and a history of approximately 15 years, the STD bus allows assembly of systems which will be maintainable for years to come.

At some sacrifice in reliability and maintainability of the system, non-critical portions, such as a graphical user interface, may be relegated to an IBM-PC/BIOS/MS-DOS platform which communicates with the embedded system via a serial link. Such a compromise may be necessary to satisfy customer demands within a reasonable time frame while staying within budget. To insure against downtime in case the PC-hosted interface goes south, the embedded software can be written to accommodate a substitute interface, such as a video terminal, a printing terminal, or a laptop running a terminal emulation program.

Training Wheels

As promised, the accompanying schematic [pages 40–41] and parts list document a reproducible, low-cost single-board computer (SBC) which will serve as a trainer for our investigation of metacompilation and embedded programming techniques.

Some readers may prefer to utilize one of the numerous 8051-family SBCs advertised in computer and electronics magazines. The advantage of the SBC presented here is that it provides for software development apart from use of EPROM programmer or ROM emulator, while making available an unfettered 64 Kbyte address space. (Well, *almost* unfettered. The uppermost eight bytes of the address space are dedicated to a parallel interface.)

The Scheme of Things

The 8051 family supports what is termed a *Harvard architecture* in which the 64K read-only code space (ROM) and the 64K read/write data space (RAM) are distinct. The spaces may, however, be combined into a single 64K region by means of external hardware. In our SBC, a pair of 32K RAMs are mapped from 0000 to FFFF in the data space, and a single ROM is mapped starting at 0000 in the code space. Processor port pin P1.2 ("switch enable" line SWEN) and the 74153 data selector control the memory map. Line PSEN- is the processor read strobe for external ROM; line RD- is the processor read strobe for external RAM. Note that a signal name designates the active (i.e., asserted) state; thus, PSEN- is low when asserted, whereas SWEN is high.

The ROM is accessed only while RDP- is low, which occurs only when the data selector routes SWEN- to RDP- while SWEN- is asserted. (Note that asserting SWEN asserts SWEN-.) SWEN- is routed to RDP- only when PSEN- is low and RD- is high; this combination of PSEN- and RD- also routes SWEN to RDM- but, if SWEN is asserted, the active-low output enables of the RAMs and the active-low read enable of the 8255 do not respond. Thus, a processor read-from-ROM instruction accesses code space of our SBC only if P1.2 is high.

If SWEN and PSEN- are low and RD- is high, then SWEN is routed through the data selector and RDM- is asserted, so that processor read-from-ROM instructions access the data space of our SBC. In this manner, code and data spaces are effectively combined into a single 64 Kbyte address space. Note that processor read-from-RAM instructions assert RD- rather than PSEN-, with the result that the data selector routes VCC (logic high) to RDP- and GND (logic low) to RDM-. Thus, the output enables of the RAMs and the read enable of the 8255 are asserted, whereas the output enable of the ROM does not respond.

Since 8051 instructions which write to memory do not assert PSEN- (i.e., writes are possible only to data space), it is a simple matter to create a routine which, while running from ROM in code space, downloads to RAM in data space.

Upon power-up, processor port pins are high. Thus, the power-up entry point is address 0000 in ROM. Initialization code in ROM can branch to a simple monitor which provides serial download capability. Once download is complete, the downloader need only write a zero to port pin P1.2 in order to transfer execution from ROM to RAM. Note that, with this scheme, the entry point for code in RAM is *not* 0000; rather, it is the RAM address *equal to* the ROM address following the instruction which writes a zero to port P1.2.

To Wrap or Not to Wrap, That is the Question

Doubtless there are a few brave souls who will, as did the author, undertake construction of a wire-wrapped version of the beast. The best prophylactic available to the wire-wrap builder is a board which provides a ground plane and is liberally furnished with bypass capacitors. Connect a 0.1 mf ceramic bypass capacitor directly between the VCC and ground terminals of each IC socket; keep the leads short. Distribute over the board several electrolytic capacitors (preferably tantalum, in the range 1 to 50 mf) between the VCC bus and ground. Verify capacitor polarities before applying power.

The less adventuresome will welcome the efforts of FIG member Ed Sisler of Santa Cruz, California who is designing a printed circuit board for the device. He plans to offer any combination from a bare board to an assembled and tested unit. Ed also can provide EPROMs containing the downloader. If there is sufficient interest, he has expressed willingness to design and market an inexpensive ROM emulator. Ed can be reached through the author, or on GEnie (address E.SISLER).

A Shopping List

When shopping for parts, note that loading problems can result unless the specified logic family (i.e., HC or HCT) is used; make substitutions only if you understand the consequences. The crystal can be virtually any 11.0592 MHz rock, regardless of rated load capacitance and design circuit configuration (i.e., series resonance or parallel resonance). C1 and C2 are disk ceramics. Bypass capacitors C3 through C8 can be either ceramic disk or multilayer ceramic. Discrete resistors can be used instead of resistor packs; the smallest typically available are 1/8 watt, which is more than ample. D2 can be a garden-variety LED; it serves as a convenient diagnostic tool to facilitate software development. Q1 isolates port 1 from the loading effects of D2. D1 can be a garden-variety switching diode. D3 provides reverse-polarity protection, and can be a garden-variety power diode. C8 is to guarantee stability of the 7805 regulator; for this purpose, C8 should be located within an inch of the 7805. A suitable power source is a wall-mount supply furnishing 9 to 15 volts DC at a few hundred milliamperes (the 7805 requires an input of at least 7 volts). The author's prototype draws less than 100 mA.

The slowest commonly available EPROMs have a maximum access time of 250 nanoseconds; this is more than adequate for an 8051 running at or below 12 MHz. Although a 32K x 8 (27256) EPROM is shown on the schematic (U9), an 8K x 8 (2764) can be used by a simple wiring change: disconnect A14 from pin 27 of U9 and tie pin 27 to VCC. EPROMs should be CMOS (i.e., 27C256 or 27C64).

The DB25S connector facilitates connection to external devices, including Centronics printers. Since each pin is connected to a programmable I/O line, the parallel port is software configurable.

The bare necessities for construction and checkout are a logic probe of the $15 variety, an inexpensive ($20 to $30) VOM (volt-ohm-milliampere meter, digital or analogue, used mostly to verify polarity and continuity), a soldering iron and tip-cleaning sponge (a damp rag will serve), miniature diagonals (for cutting component leads), and a few feet of rosin-core solder. Miniature needle-nose pliers, a solder sucker, and a magnifying lens are not essential, but will be found to be more than useful.

Miscellany

Readers using another SBC as a trainer, and those working with other processor families, will have need of a ROM emulator or the combination of EPROM programmer and ultraviolet EPROM eraser. FIG member Frank Sergeant markets an inexpensive EPROM programmer (see his article in the September/October 1992 issue). The author recently purchased from Digi-Key a Dataerase II ultraviolet eraser with timer, manufactured by Walling Company of Tempe, Arizona. I am extremely pleased, both with the design and performance of the eraser and with the manufacturer's service policy. The instruction manual contains a tutorial covering EPROMs and the production of ultraviolet light

The circuitry of the trainer was designed by FIG member Mike Foley of Houston, Texas who also compiled the parts list. Mike proposed the scheme of transferring execution from ROM to RAM. The author wire-wrapped and tested the prototype.

Initialization/downloader code will be posted on GEnie, and will be available directly from the author; kindly include an SASE or a postage stamp.

The preliminaries being now complete, the next episode will begin our exploration of metacompilation. Meanwhile, contact your local Siemens, Signetics/Phillips, or Intel supplier for an 8051-family handbook.

R.S.V.P.

Russell Harris is an independent consultant providing engineering, programming, and technical documentation services to a variety of industrial clients. His main interests lie in writing and teaching, and in working with embedded systems in the fields of instrumentation and machine control. He can be reached by phone at 713-461-1618, by mail at 8609 Cedardale Dr., Houston, Texas 77055, or on GEnie (address RUSSELL.H).

[Schematic appears on next pages.]

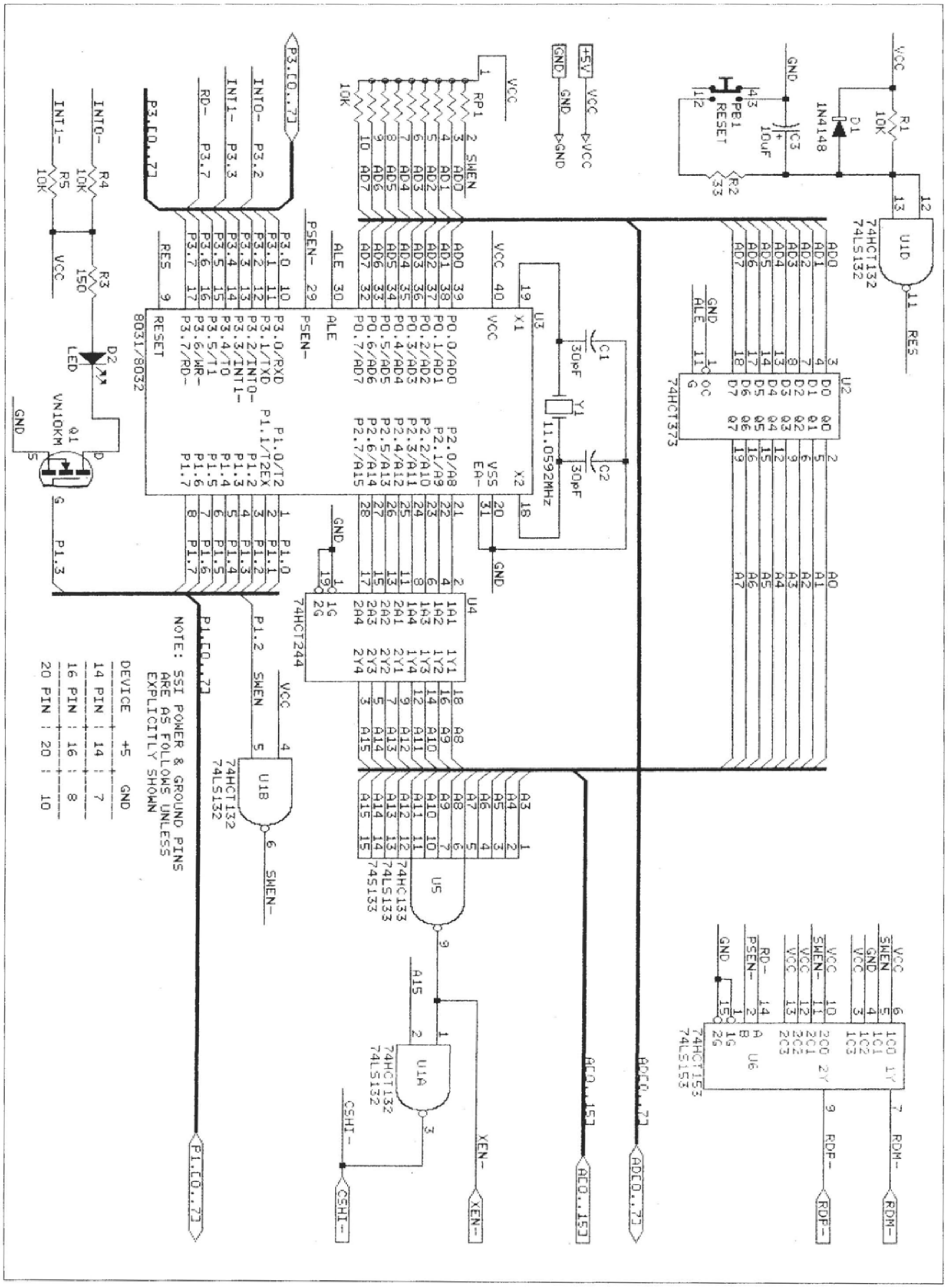
Parts list.

The following components are available from the Digi-Key Corporation, 701 Brooks Avenue South, Post Office Box 677, Thief River Falls, Minnesota 56701-0677, telephone 800-344-4539.

U1	CD74HCT132E
U2	CD74HCT373E
U4	CD74HCT244E
U6	CD74HCT153E
U5	MM74HC133N
U11	ICL232 or MAX232CPE
R1,R4,R5	10KQ
RP1	Q9103
R2	33Q
R3	150Q
Y1	X426
D1	1N4148PH
D3	1N5009PH
D2	P363
Q1	VN10LP
PB1	P8030S

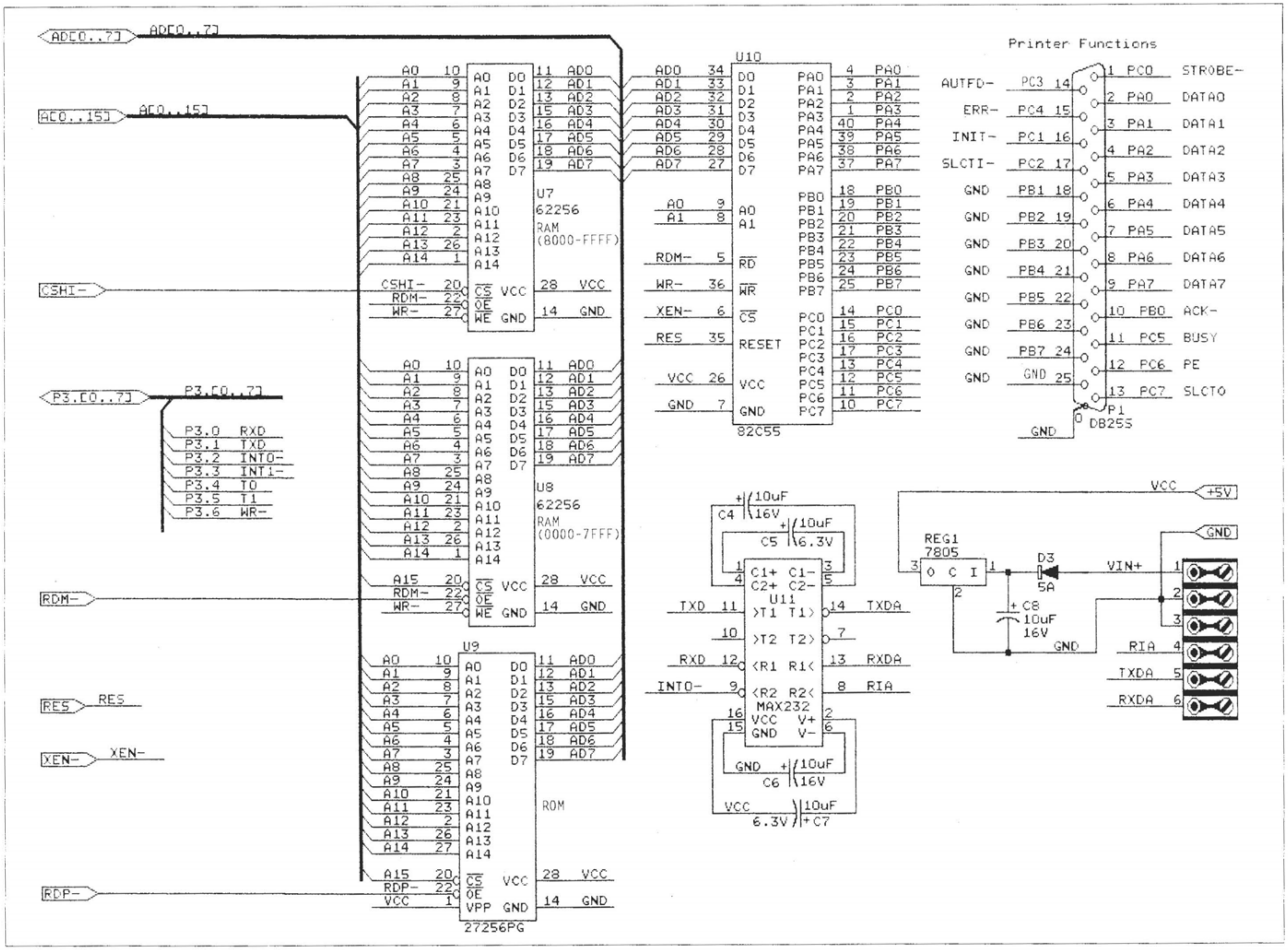
The following components are available from JDR Microdevices, 2233 Samaritan Drive, San Jose, California 95124, telephone 800-538-5000 or 408-559-1200.

U3	8031
U10	82C55-5
U7,U8	HM43256LP-10
U9	27C256 or 2764 (see text)
REG1	7805T
C1,C2	SM-30
C3,C4,C5,C6,C7,C8	T10-25
P1	DB25S



NOTE: SSI POWER & GROUND PINS
ARE AS FOLLOWS UNLESS
EXPLICITLY SHOWN

DEVICE	+5	GND
14 PIN	14	7
16 PIN	16	8
20 PIN	20	10


AD[0..7]
A[0..15]
CSHI-
P3.[0..7]
P3.0 RXD
P3.1 TXD
P3.2 INT0-
P3.3 INT1-
P3.4 T0
P3.5 T1
P3.6 WR-
RDM-
RES
XEN-
RDP-
U7
62256
RAM (8000-FFFF)
U8
62256
RAM (0000-7FFF)
U9
ROM
27256PG
U10
82C55
Printer Functions
AUTFD-
ERR-
INIT-
SLCTI-
GND
STROBE-
DATA0
DATA1
DATA2
DATA3
DATA4
DATA5
DATA6
DATA7
ACK-
BUSY
PE
SLCTO
P1
DB25S
U11
MAX232
C4 10uF 16V
C5 10uF 6.3V
C6 10uF 16V
C7 10uF 6.3V
TXD
RXD
INT0-
TXDA
RXDA
RIA
VCC
+5V
GND
REG1
7805
D3
5A
VIN+
C8 10uF 16V

HARVARD SOFTWORKS

NUMBER ONE IN FORTH INNOVATION

(513) 748-0390 P.O. Box 69, Springboro, OH 45066

MEET THAT DEADLINE ! ! !

- Use subroutine libraries written for other languages! More efficiently!
- Combine raw power of extensible languages with convenience of carefully implemented functions!
- Faster than optimized C!
- Compile 40,000 lines per minute! (10 Mhz 286)
- Totally interactive, even while compiling!
- Program at any level of abstraction from machine code thru application specific language with equal ease and efficiency!
- Alter routines without recompiling!
- Source code for 2500 functions!
- Data structures, control structures and interface protocols from any other language!
- Implement borrowed features, more efficiently than in the source!
- An architecture that supports small programs or full megabyte ones with a single version!
- No byzantine syntax requirements!
- Outperform the best programmers stuck using conventional languages! (But only until they also switch.)

HS/FORTH with FOOPS - The only full multiple inheritance interactive object oriented language under MSDOS!

Seeing is believing, OOL's really are incredible at simplifying important parts of any significant program. So naturally the theoreticians drive the idea into the ground trying to bend all tasks to their noble mold. Add on OOL's provide a better solution, but only Forth allows the add on to blend in as an integral part of the language and only HS/FORTH provides true multiple inheritance & membership.

Lets define classes BODY, ARM, and ROBOT, with methods MOVE and RAISE. The ROBOT class inherits:

```
   INHERIT> BODY
   HAS> ARM RightArm
   HAS> ARM LeftArm
```

If Simon, Alvin, and Theodore are robots we could control them with:

```
Alvin 's RightArm RAISE          or:
+5 -10 Simon MOVE                or:
+5 +20 FOR-ALL ROBOT MOVE
```

The painful OOL learning curve disappears when you don't have to force the world into a hierarchy.

WAKE UP ! ! !

Forth need not be a language that tempts programmers with "great expectations", then frustrates them with the need to reinvent simple tools expected in any commercial language.

HS/FORTH Meets Your Needs!

Don't judge Forth by public domain products or ones from vendors primarily interested in consulting - they profit from not providing needed tools! Public domain versions are cheap - if your time is worthless. Useful in learning Forth's basics, they fail to show its true potential. Not to mention being s-l-o-w.

We don't shortchange you with promises. We provide implemented functions to help you complete your application quickly. And we ask you not to shortchange us by trying to save a few bucks using inadequate public domain or pirate versions. We worked hard coming up with the ideas that you now see sprouting up in other Forths. We won't throw in the towel, but the drain on resources delays the introduction of even better tools that could otherwise be making your life easier now! Don't kid yourself, you are not just another drop in the bucket, your personal decision really does matter. In return, we'll provide you with the best tools money can buy.

The only limit with Forth is your own imagination!

You can't add extensibility to fossilized compilers. You are at the mercy of that language's vendor. You can easily add features from other languages to **HS/FORTH**. And using our automatic optimizer or learning a very little bit of assembly language makes your addition zip along as well as and often better than in the parent language.

Speaking of assembler language, learning it in a supportive Forth environment virtually eliminates the learning curve. People who failed previous attempts to use assembler language, often conquer it in a few hours using HS/FORTH. And that includes people with NO previous computer experience!

HS/FORTH runs under MSDOS or PCDOS, or from ROM. Each level includes **all** features of lower ones. Level upgrades: $25. plus price difference between levels. Source code is in ordinary ASCII text files.

HS/FORTH supports megabyte and larger programs & data, and runs as fast as 64k limited Forths, even without automatic optimization -- which accelerates to near assembler language speed. Optimizer, assembler, and tools can load transiently. Resize segments, redefine words, eliminate headers without recompiling. Compile 79 and 83 Standard plus F83 programs.

PERSONAL LEVEL $299.
NEW! Fast direct to video memory text & scaled/clipped/windowed graphics in bit blit windows, mono, cga, ega, vga, all ellipsoids, splines, bezier curves, arcs, turtles; lightning fast pattern drawing even with irregular boundaries; powerful parsing, formatting, file and device I/O; DOS shells; interrupt handlers; call high level Forth from interrupts; single step trace, decompiler; music; compile 40,000 lines per minute, stacks; file search paths; format to strings. software floating point, trig, transcendental, 18 digit integer & scaled integer math; vars: A B * IS C compiles to 4 words, 1..4 dimension var arrays; **automatic optimizer delivers machine code speed.**

PROFESSIONAL LEVEL $399.
hardware floating point - data structures for all data types from simple thru complex 4D var arrays - operations complete thru complex hyperbolics; turnkey, seal; interactive dynamic linker for foreign subroutine libraries; round robin & interrupt driven multitaskers; dynamic string manager; file blocks, sector mapped blocks; x86&7 assemblers.

PRODUCTION LEVEL $499.
Metacompiler: DOS/ROM/direct/indirect; threaded systems start at 200 bytes, Forth cores from 2 kbytes; C data structures & struct+ compiler; MetaGraphics TurboWindow-C library, 200 graphic/window functions, PostScript style line attributes & fonts, viewports.

ONLINE GLOSSARY $ 45.

PROFESSIONAL and PRODUCTION LEVEL EXTENSIONS:

FOOPS+ with multiple inheritance **$ 79.**
TOOLS & TOYS DISK $ 79.
286FORTH or **386FORTH $299.**
16 Megabyte physical address space or gigabyte virtual for programs and data; DOS & BIOS fully and freely available; 32 bit address/operand range with 386.
ROMULUS HS/FORTH from ROM **$ 99.**

Shipping/system: US: $9. Canada: $21. foreign: $49. We accept MC, VISA, & AmEx

On the Back Burner #5

A Lesson in Economics

Conducted by Russell L. Harris
Houston, Texas

Ed Zern relates the tale of the ill-fated manufacturer, Tates, which attempted to branch out into the production of magnetic compasses of the sort used by hunters, fishermen, and Boy Scouts. Unfortunately, Tates had no prior experience in the manufacture of compasses. Being unaware of the pitfalls and design subtleties inherent in this field of endeavour, Tates management poured millions into production and marketing, but almost nothing into research and engineering. In appearance, the Tates compass was a thing of beauty. It was nicely proportioned, and appeared to be well-crafted and ruggedly built. Nevertheless, for reasons now lost in the annals of history, the design was flawed. Sometimes the device worked perfectly, giving an accurate indication; other times, without warning or indication that things were amiss, the compass would give readings seriously in error. Outdoorsmen soon learned from personal experience that the Tates compass was not to be trusted, thus giving rise to the saying, "He who has a Tates is lost."

A Matter of Misfeasance

The IBM-PC family, running under PC BIOS and MS-DOS, has become perhaps the most widely misapplied component in the field of real-time instrumentation and control. Direct control of machinery by a PC is a practice fraught with many perils and considerable difficulty. Whether in ignorance or in deliberate disregard of the body of facts and documented experience presented by a multitude of authors in a variety of engineering publications, programmers and newly emerging systems houses daily foist upon their clients systems in which the PC is directly interfaced with "the real world." Often, it is not until a project has passed the point of no return that the full consequence of matters such as indeterminate interrupt latency, bugs in BIOS, and bugs in DOS become apparent.

Sometimes the PC is disguised by means of packaging. It seems that hardly a day goes by without the introduction of another PC-compatible single-board computer targeted at the embedded control market. Although such systems may utilize solid-state memory instead of a mechanical disk drive, they nonetheless possess the basic liabilities of the desktop PC.

Penny-Wise, Pound-Foolish

The IBM-PC/BIOS/MS-DOS environment lures the unsuspecting real-time user with the siren call of cheap hardware, inexpensive software, and a seeming abundance of capable programmers. Not until one makes a detailed cost analysis, taking into account system lifetime, does the true cost become apparent. Moreover, industrial experience continues to demonstrate that commodity-grade C programmers armed with mass-marketed software packages are generally inept in the realm of real time.

Aside from illusory economies in the area of hardware, the basic impetus toward the IBM-PC/BIOS/MS-DOS environment is the elusive goal of automatonistic software creation. Viewing the variety of inexpensive mass-marketed MS-DOS software, one finds appealing the possibility of creating customized software for a system simply by blindly combining a number of off-the-shelf programs. However, projects taking this approach repeatedly demonstrate that (1) the expense incurred in attempting to integrate general-purpose programs for which one is lacking the source code can easily exceed the cost of a tailor-made program, and (2) the resulting hodgepodge tends to be unreliable and full of arcane idiosyncrasies, in addition to aberrations inherent in the IBM-PC/BIOS/MS-DOS platform.

It is virtually impossible for a programmer to have complete control of a system based on the IBM-PC/BIOS/MS-DOS platform, for it is virtually impossible for a programmer to gain access to the source code for all the pertinent software and firmware modules. Moreover, the complexity of the platform precludes accurate prediction of system response to real-time stimuli. The situation is exacerbated when commercial software packages are integrated into the system.

When one commits himself to the IBM-PC/BIOS/MS-DOS platform, he enters an environment of continual change. Whether from the standpoint of hardware or that of software, the window of availability in the PC world is short compared with the service life of typical industrial systems. The high sales volume which drives down the cost of PC hardware and software both permits and encourages frequent redesign, in order to reduce manufacturing cost and to enhance performance. By comparison, common industrial components, such as relays, valves, and transducers, generally continue in production without change for dozens of years.

The problem is maintainability: after a system incorporating a PC has been in service a year or two, it may be very difficult to find replacement components which are compatible with the original PC hardware. Even if such components are available, they may not be compatible with the original software, apart from reconfiguration or modification of the software. If the original software cannot accommodate the available hardware, extensive software modification may be required. If the software has been integrated from a number of commercial software packages and one or more of these must be updated, the programmer must again deal with the same problems of software/software and software/hardware compatibility he faced in patching together the original system.

("Back Burner" continues on page 38.)

CALL FOR PAPERS

for the fifteenth annual and the 1993

FORML CONFERENCE

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

November 26 – November 28, 1993
(following Thanksgiving)
Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California U.S.A.

Theme: **Forth Development Environment**

Papers are invited that address relevant issues in the establishment and use of a Forth Development Environment. Some of the areas and issues that will be looked at consist of Networked Platform Independence, Machine Independence, Kernel Independence, Development System-Application System Independence, Human-Machine Interface, Source Management and Version Control, Help Facilities, Editor-Development Interface, Source and Object Libraries, Source Block and ASCII Text Independence, Source Browsers (including Editors, Tree Displays and Source Database), Run-Time Browsers (including Debuggers and Decompilers), Networked Development-Target Systems.

Additionally, papers describing successful Forth project case histories are of particular interest. Papers about other Forth topics are also welcome.

Mail abstracts of approximately 100 words by September 1, 1993.
Completed papers are due November 1, 1993.
We anticipate a full conference this year.
Priority will be given to participants who submit papers.

John Hall, Conference Chairman

Robert Reiling, Conference Director

Information may be obtained by phone or fax from the
Forth Interest Group, P.O. Box 2154, Oakland, California 94621. 510-893-6784, fax 510-535-1295
This conference is sponsored by FORML, an activity of the Forth Interest Group, Inc. (FIG).

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