VOLUME XII, NUMBER 2 JULY/AUGUST 1990 $6.00

FORTH DIMENSIONS

UPSCALE NUMBER INPUT

68000 NATIVE-CODE FORTH

EXTENSIBLE OPTIMIZING COMPILER

DICTIONARY STRUCTURES AND FORTH

INTERACTIVE CONTROL STRUCTURES

METACOMPILE BY DEFINING TWICE

FORTH DIMENSIONS

EDITORIAL

Warning—whether you are a moderately proficient or a profoundly immodest Forth programmer, you'll find some challenges herein. A couple of our contents-page thermometers almost shattered, but some of you wanted to sink your teeth into advanced topics...

Phil Koopman, Jr. suggested in our last issue—and plenty of working, real-world evidence supports him—that Forth is the language of choice for embedded control systems. But now Wu Qian explains Forth's underlying structure in a way that makes it seem the ideal design-and-construction kit for operating systems. Every programmer seems to have his own idea about what Forth is and what it does best. It brings to mind Kim Harris' early article in *Dr. Dobb's Journal* (1981), still the seminal description of "The Forth Philosophy" in the minds of many. It was amusing then to think that some people couldn't decide whether Forth was a language, an operating system, a way of thinking, or a metaphor.

Forth still retains a mutative, slippery kind of strength that is hard to convey to the uninitiated. That is why it's so tough to sell, and why Forth marketing pitches of the past often used an "all things to all people" line that no one took seriously. Of course we could make Forth into something more graspable and less elusive; but that, we suspect, is no longer Forth. We are still unable to say in meaningful specifics just what Forth is, unable to agree on any less-than-global description.

Phil's idea, applied expertly, could bring Forth into the spotlight. NASA's Douglas Ross evidently agrees, for he wrote a vociferous letter to *Electronic Design* in response to an editorial titled, "Embedded Programming: C or Ada?" that only passingly mentions Forth and Pascal. As Douglas points out, "Forth, for those who don't know, was created in the early 1970s by Charles Moore to specifically address the needs of programming control applications, in an interactive and efficient manner. Forth has been *the language of choice* 'embedded' in processors to control video games, washing machines, calculators, radio telescopes, and satellite control systems. It has also been written to run on probably the widest array of processors known." If you have access to back issues of that magazine, see the complete letter in the May 11, 1989 issue. (The added italics are mine, in reference to the Koopman article.)

Mr. Ross also has a letter in this issue of *Forth Dimensions*, but related to the Sieve algorithm. Finding prime numbers is one of those programmer's perennials, it always seems to elicit reader response and rebuttal. Check our letters section for the latest on the topic and some interesting code.

—*Marlin Ouverson*
Editor

Reviewers' Remarks

We are pleased to bring to our pages another prize-winning Forth author. Andrew Scott's "Extensible Optimizer for Compiling Forth" won the award for Best Paper at the 1989 FORML Conference in Monterey, California. He and John Redmond, whose ultimately optimized *tour de ForST* is featured in this issue, each present an optimizable, subroutine-threaded Forth. Unlike Redmond's work on the ST, Scott's is meant to be ported to many different microprocessors. Both authors present us with some original thinking, work showing that Forth does not have to suffer in terms of speed compared to other languages. These may turn out to be breakthrough developments; readers are encouraged to study them and respond.

Forth Dimensions
Published by the
Forth Interest Group
Volume XII, Number 2
July/August 1990
Editor
Marlin Ouverson
Advertising Manager
Kent Safford
Design and Production
Berglund Graphics

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 $30 per year ($42 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 8231, San Jose, California 95155. Administrative offices and advertising sales: 408-277-0668.

About 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 $24/36 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 8231, San Jose, CA 95155."

LETTERS

Poly-Lingual FIG?

In the Forth community, speaking about other languages isn't a taboo (maybe because each of us is using his own Forth extension or Forth-like language). So usually Forth people are curious about other languages; for one, I would like to include any good idea in my Forth system. It might be good for the Forth Interest Group to offer documentation about any existing language, especially those for which it is difficult to find such details. At various times, Forth articles refer to Neon, Reptil, PISTOL, Stoic, Sphere, Magic-L, etc., but I don't know where to find documentation.

Another idea is to extend the name of the Forth Interest Group to FLIG—Forth & Languages Interest Group. The Forth community is a very good multi-language forum; why don't we institutionalize that fact?

Giorgio Kourtis
Via Ameglia 1/9
16136 Genova
Italy

A Smaller Prime Thousand...

Our last issue contained a letter titled "Fast Thousand Seems Slow," in which Marc Hawley presents his improvement to the Eratosthenes Sieve code printed with an earlier letter from Allan Rydberg (FD XI/5). The following writer makes the same improvements, but in a different dialect and with subtle differences that Sieve sifters may find of interest.

Dear Sirs,

....I will describe an optimized version of the original [Sieve]... The algorithm finds all prime numbers smaller than a certain limit, rather than finding a given number of primes. To each integer up to this limit, we store a flag. We then proceed to eliminate all products of prime numbers: starting with 2 as our prime, we mark in our array the numbers 4, 6, 8, ... as unprime. Now we eliminate all products of three: 6, 9, 12, ... Examining our array, we find 4 is unprime, so there is no need to eliminate its products, these having already been eliminated by one (or more) of its prime factors (in this case, 2). Continuing, we eliminate 10, 15, 20, ..., then 14, 21, 28, ..., etc. We observe, however, that we can begin eliminating products of any prime number p at p^2, since any product of p which is smaller than p^2 must also be a product of a lower prime, and will thus already have been eliminated. We end the process when the square of the prime we are about to use for our eliminations is larger than our limit. All we now have to do is print the prime numbers.

I enclose a short (two screens) Forth implementation of the algorithm, written for clarity rather than speed. It is written in a Forth-79/fig-FORTH hybrid, but does not contain any irregular features. All primes up to 10,000 are found and printed. Since there are 1230 primes up to this limit, comparisons can be made. On my 2 MHz 6502 system (a BBC computer running

Listing One. Scolnicov's smaller thousand.

```
SCR # 7      7 H
( Eratosthenes' Sieve Program by Ariel Scolnicov )
10000 CONSTANT NPRIMES
CREATE PRIMES NPRIMES 8 / 1+ ALLOT
HEX
CREATE BIN-TBL 0102 , 0408 , 1020 , 4080 ,
DECIMAL
: 2^ BIN-TBL + C@ ;
: SET ( i -- ) 8 /MOD PRIMES + DUP C@ ROT 2^ OR SWAP C! ;
: GET ( i -- bit-nonzero ) 8 /MOD PRIMES + C@ SWAP 2^ AND ,
: ELIM  ( factor -- ; eliminates all products of factor )
   DUP ." Eliminating products of " . ." ..." CR
   DUP DUP * NPRIMES SWAP  ( factor from to )
   DO  I SET DUP +LOOP DROP ;

SCR # 8      8 H
( SIEVE scr #2 )
VARIABLE PRIMES-FOUND
: SIEVE     ( -- ; sieves table )
   2 BEGIN  DUP GET NOT  IF DUP ELIM THEN
      DUP DUP * NPRIMES < WHILE 1+ REPEAT DROP ;
   PRIMES  0 PRIMES-FOUND !  NPRIMES 1    DO
     I GET NOT IF I 8 .R 1 PRIMES-FOUND +! THEN LOOP ;
: INIT PRIMES NPRIMES 8 / 1+ 0 FILL ;
: FIND-PRIMES INIT SIEVE .PRIMES CR ;
```

Acornsoft FORTH), the process takes 114 seconds, with half of this time used by `.PRIMES` and only 57 seconds for finding the primes. Doubtless, the program will run much faster on the Amiga. Memory used is 1623 bytes, since only one bit is used to represent each number. The running time for the program grows as the square root of the limit number, compared with linear times for Mr. Rydberg's program. However, storage space does grow linearly in my program, while Mr. Rydberg's grows far less.

Finally, a few comments on Mr. Rydberg's comments:

- Division on a computer is carried out in exactly the way he describes, except that the quotient is also calculated on the way. A primitive `MOD` should, therefore, be faster than his test.
- To find the squares of the primes in a different way, utilize the congruences $1^2 = 1, 2^2 = 1+3, 3^2 = 1+3+5, \ldots, n^2 = 1+3+\ldots+2*n-1$. Thus, to find the square of the newly found prime p, use the square of the previous prime q: $p^2 = q^2 + (2*q+1) + (2*q+3) + \ldots + (2*p-1)$. It is unclear, however, if this will be faster for large primes, since they are widely dispersed.
- In testing the integers one-by-one for primeness, only integers of the form $6n\pm1$ need to be tested, since any other remainder after division by 6 means the number is divisible by 2 or 3. (Thanks to my younger brother for this.)

Yours,
Ariel Scolnicov
Nof Harim 96, Box 2747
Mevasseret Zion
Israel 90805

...and the Fastest (?) Thousand

Dear Sir,

In the November 1989 edition of *BYTE*, Milton Pope's letter presents a fast Sieve algorithm in BASIC based on the work presented by Nick Pelling in a letter from the May 1989 issue of that magazine. He showed two versions of the algorithm and suggested a third, even more efficient, algorithm.

Presented here is a fast Sieve using all the ideas suggested by Mr. Pope. It uses only prime numbers to test multiples against, starting at the square of the prime, up to the square root of the array being searched.

The array represents only the odd numbers (except for index 0, which represents the prime 2). Therefore, you can find all the primes in value up to two times the array size. The index values 1, 2, 3, 4,... correspond to the numbers 3, 5, 7, 9,...

`PRIMES` computes and displays the number of primes found in the array. `.PRIMES` displays the prime values (ten to a line). `SIZE .PRIMES` displays all the prime values from 2 through `2*SIZE`.

Listing Two. Ross' fastest (?) thousand.

```
\ Fastest (?) Eratosthenes Sieve Benchmark
\ Douglas Ross, NASA GSFC, 12/6/89
\ Computes the primes from 2 to 2*SIZE
\ There are 1028 primes up to 8190, 1899 up to 16380

 DECIMAL
 8190 CONSTANT SIZE    90 CONSTANT ROOT   CREATE FLAGS SIZE ALLOT
 : DO-PRIME
   FLAGS SIZE 1 FILL                     \ initialize array
   ROOT 1                                \ do ROOT times
   DO FLAGS I + C@                       \ flags[i]  --
      IF I 2* 1+ DUP 2* SWAP DUP *       \ step  prime^2  --
         BEGIN  DUP SIZE 2* U<           \ step  prime^2  ?  --
         WHILE  0 OVER 2/                \ step  prime^2  0  i  --
                FLAGS + C!               \ step  prime^2  -- | flag[i] = 0
                OVER  +                   \ step  prime^2+step[i] --
         REPEAT DROP DROP                \ --
      THEN
   LOOP ;


The following words are for outputting information.

  : PRIMES 1 SIZE 1 DO FLAGS I + C@ + LOOP . ." PRIMES " ;

  VARIABLE CNT
  : 10=  CNT @ 1+ DUP 10 = IF 0 CNT ! CR ELSE CNT ! THEN ;

  : .PRIMES ( SIZE -- ) CR  2 8 U.R  1 CNT !

     1 DO FLAGS I + C@ IF I 2* 1+ 8 U.R 10= THEN LOOP ;
```

Yours truly,
Douglas Ross
NASA GFC
Code 728
Greenbelt, Maryland 20771

Macro Update

Dear Marlin,

The article "Macro Generation" by Don Taylor (*FD* VII/1) is labelled as Forth-83 but its definitions wouldn't work on a Forth-83 system. It uses `TIB` as if it were a variable, and doesn't use `#TIB` to set the length of the console input stream. I guess this is a little late to report errors, but the macro generator seems quite clever and worth using sometimes to cut away the nest-unnest overhead of short colon definitions and to help make source code more readable. Screens one through four here hold a definition that works on my Forth-83 system. I renamed `TIB!` to `!TIB` to more clearly show that its function is to restore something, not memory access.

This version of `MACRO:` is an extension of the normal interpretation of the input stream. It implicity assumes that, after its compiled string is installed as the new input stream, control immediately returns to words that interpret (and execute or compile) the input stream. Its macros can't be performed within a colon definition. They could be invoked within a program, but would not be executed until (and if) the program ended and gracefully returned control to the console.

If `INTERPRET` is available, interpreted tasks can be called from within an application. Screen five holds a definition that does so. The only use that occurs to me for such a run-time macro is to use `FORGET` <name> to trim startup words from an application and still be able to use `FIND` and `'` to locate words. (Calling `ALLOT` with a negative number trims the dictionary but destroys the dictionary chains.)

Sincerely,
David Arnold
616 1/2 W. Hamilton Street
Kirksville, Missouri 63501

Hindsight

A gremlin struck "Stack Variables" in the last issue on page 21. With our apologies, the first line of code in Figure One should be:

```
: INCLUDE   ( filename$ -- )
```

Listing Three. Arnold's macro update.

```
Screen# 1        Forth-83
( interpreted macro definition                    13Apr90dna )
only  forth  definitions  also   decimal
create MACRO-MARK                           ( FORGET'able marker )
: C,  ( c -- )  here  1 allot  c! ;
: !TIB  ( -- )                   ( Can restore TIB if macro bombs. )
 [ tib ] literal  [ ' tib >body ] literal !
 ." TIB repaired. " cr  quit ;
 3  load                ( immediate interpretation )
( 5  load                ( run-time interpretation )
only  forth  definitions  also   decimal
( Adapted from article by Don Taylor )
( pp. 27-28, 'Forth Dimensions' vol. 7, no. 1. )
```

```
Screen# 2        Forth-83
( macro - immediate interpretation                13Apr90dna )
-- source format --
MACRO: <name>  <source_text> ... ;
Put a 'spc' between the macro text and the trailing ';'.
-- compilation & execution --
Compile macro text & a trailing '|' as a counted string.
Leave macro text area long enough for possible '!TIB' entry.
  During interpretation (or compilation) of input stream, save
  state of input stream & redirect it to compiled text, which is
  treated as if it were a console entry.
  '|' at end of macro restores state of input stream.
-- fault recovery --
!TIB <enter>  can restore the TIB pointer if a macro bombs.
Don't put a 'spc' before or after !TIB, because at most 4 bytes
are guaranteed in a macro text area.  Then, FORGET the macro,
which would have been corrupted by the console entry.
```

```
Screen# 3        Forth-83
( macro - macro compilation & execution           13Apr90dna )
             ( Appended to macro text to restore input stream. )
: |  ( --   #tib tib >in blk -r- )
 r> ( IP )
 r> r> r> r>   #tib !  [ ' tib >body ] literal !  >in !  blk !
 >r ;   immediate
: MACRO:  ( -- )
 create  immediate
 59 ( ';' ) word   count  dup 1+  dup c,   rot rot
 ?dup if   0 do  count c,  loop   then   ( len+1 txt len -- .. )
 drop   124 ( '|' ) c,                   ( .. -- len+1 )
 4 swap - 0 max  allot        ( Ensure space for a !TIB entry. )
 does>  ( .. str --   -r- #tib tib >in blk )
   r> ( IP )   blk @  >in @  tib  #tib @  >r >r >r >r   >r
   count
   #tib !  [ ' tib >body ] literal !  0 blk !  0 >in ! ;
```

```
Screen# 4        Forth-83
( macro - run-time macro interpretation           13Apr90dna )
-- source format --
MACRO: <name>  <source_text> ... ;
Put a 'spc' between the macro text and the trailing ';'.
-- compilation & execution --
Compile macro text as a counted string.
Leave macro text area long enough for possible '!TIB' entry.
  Save state of input stream, call INTERPRET to execute macro as
  if it were a console entry, then restore input stream.
-- fault recovery --
!TIB <enter>  can restore the TIB pointer if a macro bombs.
Don't put a 'spc' before or after !TIB, because at most 4 bytes
are guaranteed in a macro text area.  Then, FORGET the macro,
which would have been corrupted by the console entry.
```

```
Screen# 5        Forth-83
( macro - macro compilation & execution           13Apr90dna )
: MACRO:  ( --   MACRO: <text> ; )
 create
 59 ( ';' ) word   count  swap  over
 dup c,
 ?dup if   0 do  count c,  loop   then     ( len txt len -- .. )
 drop
 4 swap - 0 max  allot     ( Ensure space for a '!TIB' entry. )
 does>  ( .. str -- )
   blk @  >in @  tib  #tib @
   >r >r >r >r
   count   #tib !  [ ' tib >body ] literal !  0 blk !  0 >in !
   interpret
   r> r> r> r>
   #tib !  [ ' tib >body ] literal !  >in !  blk ! ;
```

UPSCALE NUMBER INPUT

GLENN LINDERMAN - SANTA CLARA, CALIFORNIA

This article was inspired by Mike Elola's article in *Forth Dimensions* (XI/4). I liked the overall concept of the ideas presented in his article, but found some points that were insufficient for my applications. The accompanying code adjusts for these insufficiencies. A comparison of Mike's implementation and mine is at the end of the article.

In addition to number input, this package includes formatted number output, number printing, and number-to-formatted-string conversions. For those who implement this package on their system, it comes complete with some test cases at the end of the code, which you can run to validate the correctness of your implementation and to inspire you with ideas for making your own picture strings.

It is better to ... conform to common user practice.

Features

The particular features provided by this implementation center around a picture string that describes the desired format of a number. The same picture string that is used to display a number can be used during the input of the number, ensuring that the number does not exceed the bounds of the picture string. Using a picture string for number input permits the number input routine to be more user friendly: for example, if a dollars-and-cents value is expected, a dollars-and-cents template is displayed during input. This is generally more helpful than describing, in a prompt, that a dollars-and-cents value is expected.

```
\ This code written for a 32-bit FORTH system that closely adheres to the
\ FORTH-83 standard.  The following extensions are used in this program:

\ All numeric operators deal with 32-bit quantities. Other FORTH systems may
\ require the use of double-number words to achieve the same data range.

\ The "case" syntax used is from the Wizard of Costa Mesa, Wil Baden.
\ Reference is Forth Dimensions Volume 8, Number 5, Page 29.

\ The words "for" and "next" were borrowed from Chuck Moore, who invented them
\ for the Novix chip. For those without them, I have used them only in ways
\ where it is equivalent to substitute "0 do" for "for", and "loop" for "next".

\ The words "2>r" and "2r>" are equivalent to ">r >r" and "r> r>",
\ respectively.

\ You may need some of these definitions if you don't have them:
\ : 4/ 4 / ;
\ : u>= swap u< ;
\ : u> u< 0= ;
\ : << for 2* next ;
\ : swapdrop swap drop ;
\ : beep 7 emit ; \ this is machine dependent
\ : us>d 0 ; \ "unsigned single to double" this is machine dependent

\ My compiler recognizes numbers beginning with $ as being hex. If yours does
\ not, the $ can be eliminated by using the decimal equivalent.

\ The stack notation used is conventional if it fits on a single line, but
\ unconventional if it doesn't. Multi-line stack notation is equivalent to
\ the single line notation, and the use of a standard text editor "join lines"
\ function can convert from multi-line notation to (very long) single line
\ notation.  Stated another way, the logical top-of-stack appears at the
\ bottom of the notation. This notation was suggested by the reviewers, to
\ clarify long stack notations.

\ You can pick your own values for these terminal control keys:
\ (Those listed form the WordStar diamond for standard ASCII QWERTY keyboards.)
19 constant cursorleft
 4 constant cursorright
 5 constant cursorup
24 constant cursordown

\ limit values (these are for a 32 bit implementation)
$7fffffff constant maxpositive
$ffffffff constant maxunsigned
$80000000 constant maxnegative

\ determine the maximum legal value for various signs.
: ni~max ( sign -- max 1/20/90 )
```

```
case 3 > if 4/ 1 swap for base @ * next 1- endcase
case 0= of maxunsigned endcase
case 1 = of maxpositive endcase
drop maxnegative ;

\ insert a digit into value at the position given.
: ni~adddigit ( value
             position
                 sign
                digit
                   --
       adjusted_value
             position
                 sign
                 0|-1  1/20/90 )
 \ first we bounds check to avoid overflow:
 \  all is well if: MAX r - position / digit - base / value position /  >=
 \   where  r = value % position
 \          v = value / position
 \          v = v * base + digit
 \          adjusted_value = v * position + r

 \ do the checking
 over ni~max 4 pick us>d 5 pick um/mod drop - us>d 4 pick um/mod swapdrop
  over - us>d base @ um/mod swapdrop
 4 pick us>d 5 pick um/mod swapdrop u< if drop -1 exit then

 \ do the work
 swap >r -rot >r us>d r@ um/mod base @ * rot + r@ * + 2r> 0 ;

\ perform error checking and return value adjustment after ni~adddigit
: ni~adderrchk ( 0|-1 original_key -- original_key|0  1/20/90)
 over if swap then drop ;

\ delete a digit from value at the position given.
: ni~deldigit ( value
             position
                 sign
                   --
       adjusted_value
             position
                 sign  1/20/90 )
 2>r us>d r@ um/mod us>d base @ um/mod swapdrop r@ * + 2r> ;

\ handle negation of the number, if within bounds
: ni~negate ( value
           position
               sign
                key
                 --
              value
  adjusted_position
      adjusted_sign
              key|0  1/20/90 )
 \ - key attempts to change the sign of the number
 \ + key attempts to make the number positive
 over 3 and if \ sign characters are permitted only if processing signed
numbers
  dup ascii - = if
   over 2 xor
  else
   over -3 and
  then \ now we have desired sign
  dup ni~max 5 pick u>= if >r 2drop r> 0 else drop then
 then ;

\ increase position value to simulate cursor left command
```

After the template has been displayed, the user can enter the number using the digit keys (which, for some number bases, includes some of the alphabetic keys), the cursor control keys (which are implementation defined), and the following characters:

-	Tells `numin` to negate the number.
+	Tells `numin` to take the absolute value.
backspace	Tells `numin` to delete the digit to the left of the cursor.

The cursor control keys have the following effects:

- ← Moves the cursor one digit to the left.
- → Moves the cursor one digit to the right.
- ↑ Adds one to the digit to the left of the cursor.
- ↓ Subtracts one from the digit to the left of the cursor.

The picture-string characters that are implemented by this code are as follows:

- 0 Represents a digit position, displayed as a zero if no more significant digits remain in the number being displayed.
- 9 Represents a digit position, displayed as a space if not significant.
- 8 Represents a digit position, displayed as an "_" if not significant.
- 7 Represents a digit position, displayed as an "*" if not significant.
- $ Leftmost $ represents a floating-position $, which can float to the rightmost insignificant $ position. Subsequent $s represent digit positions, and the second and subsequent insignificant $ (from the right) are displayed as spaces. This only sounds complicated, in an attempt to be precise; generally speaking, a row of $ causes a single $ to be placed adjacent to the most significant digit.
- 1 Represents zero or more digit positions, exactly enough to display all remaining significant digits.
- 2 Represents zero or more digit positions, exactly enough to display all remaining significant digits, with an embedded comma every three digits.
- , represents a comma if there are significant digits to its left; if there are not, it displays as whatever the next

(Screens continued on next page.)

digit position to the right would have displayed as if it were not significant. Resets the comma counter for implicit commas for the picture code 2.

. Displays as ".". Resets the comma counter for implicit commas for the picture code 2.

For signed numbers, additional picture codes are allowable. They are:

- \- Displays as "-" for negative numbers, space for non-negative numbers.
- \+ Displays as "-" for negative numbers, "+" for non-negative numbers.
- (Displays as "(" for negative numbers, space for non-negative numbers.
-) Displays as ")" for negative numbers, space for non-negative numbers.
- 3 Displays as "CR" for negative numbers, two spaces for non-negative numbers.
- 4 Displays as "DB" for negative numbers, "CR" for non-negative numbers.
- 5 Displays as "CR" for negative numbers, "DB" for non-negative numbers.
- 6 Similar to $ above, but implements a floating "-" picture code. One position coded as a 6 will contain the sign indication, positions to the left will be blank, positions to the right will contain digits.

Any other characters that might appear in a picture string are copied to the output string.

The multitude of different techniques for displaying the sign reflects the multitude of commonly used techniques used by different groups of accountants. It is much better to have the capability to conform to common user practice than to require that the user learn new techniques, so the whole multitude was implemented.

Implementation

The cursor is permitted to be positioned only at digits. Rather than scan forward or backward in the picture string for the next digit position (which could be rather hard for picture codes 1 and 2, which represent multiple digits), the cursor position is maintained by keeping track of the position of the cursor in mathematical terms—the "position" is given by the base raised to a power: the power zero represents the units position, the power

```
\ permitted if:  value base @ / position >= value position > and
: ni~incrpos (value position sign -- value adj_position sign flag 1/20/90 )
 >r 2dup u> if
   r@ ni~max us>d base @ um/mod swapdrop over u>= if
     base @ * false
   else
     true
   then
 else
   true
 then r> swap ;

\ decrease position value to simulate cursor right command
: ni~decrpos (value position sign -- value adj_position sign flag 1/20/90 )
 >r dup base @ u>= if base @ / false else true then r> swap ;

\ increase value by 1 at current digit position
: ni~incrdigit ( value
              position
                  sign
                    --
        adjusted_value
              position
                  sign
                  flag  1/20/90 )
 3dup ni~max rot - u> if true else 2>r r@ + 2r> false then ;

\ decrease value by 1 at current digit position
: ni~decrdigit ( value
              position
                  sign
                    --
        adjusted_value
              position
                  sign
                  flag  1/20/90 )
 over2 over2 u>= if 2>r r@ - 2r> false else true then ;

\ The following number input helper word does initial processing on each
\ keystroke. Because most of the keystrokes are expected to affect the value
\ of the number, this routine is called first to analyze each keystroke and
\ apply it to the current numeric value. If the keystroke has no effect on
\ the number, it is left on the stack.  If this routine uses the keystroke,
\ the value and sign on the stack are updated, and the keystroke replaced by
\ a zero. One implication of this behavior is that the NUL character is
\ ignored. This behavior seemed permissible in this context.  If that is
\ unacceptable in your application, you can either pre-check for NUL before
\ calling this word, or define a meaning for NUL within this word.  This word
\ is sensitive to the number base, and will permit as digits only those
\ characters from 0-9, A-Z, and a-z that are permitted by the current number
\ base.  Support is provided only for number bases up to 36, base 36 is
\ substituted for larger bases.

: ni~keyadj ( value
           position
               sign
           key_code
                 --
     adjusted_value
  adjusted_position
      adjusted_sign
         key_code|0  1/20/90 )

 case ascii - = ascii + =or
\ possible sign adjustment
    if ni~negate endcase
```

```
 case 8 = 127 =or
\ BS, DEL delete last digit
   of ni~deldigit false endcase

 case ascii 0 - 10 base @ min 0 max u<
\ digit from 0 to 9 (within base bounds)
   if dup>r ascii 0 - ni~adddigit r> ni~adderrchk endcase

 case ascii A - 26 base @ 10 - 0 max min u<
\ digit from A to Z (within bounds)
   if dup>r ascii A - 10 + ni~adddigit r> ni~adderrchk endcase

 case ascii a - 26 base @ 10 - 0 max min u<
\ digit from a to z (within bounds)
   if dup>r ascii a - 10 + ni~adddigit r> ni~adderrchk endcase

 case cursorleft =
\ increase position for cursor left
   if >r ni~incrpos r> ni~adderrchk endcase

 case cursorright =
\ decrease position for cursor right
   if >r ni~decrpos r> ni~adderrchk endcase

\ the following two cases implement absolute value increase and decrease.
\ if you wish to implement actual value increase and decrease, you could
\ check the sign and call either ni~incrdigit or ni~decrdigit as needed.

 case cursorup =
\ increase digit value
   if >r ni~incrdigit r> ni~adderrchk endcase

 case cursordown =
\ decrease digit value
   if >r ni~decrdigit r> ni~adderrchk endcase

 \ no more cases: return key_code without adjustment
 ;

: no~bumpcurs ( cursctrl -- adjusted_cursctrl  1/20/90 )
 dup 65535 and 255 > if 1+ then ;

: no~bumpcursdigit ( cursctrl -- adjusted_cursctrl  1/20/90 )
 65536 + dup 65535 and 255 > if 255 - then ;

: no~dispcurs ( cursctrl --  1/20/90 )
 255 and dup 1 > if 1- for 8 emit next else drop then ;

: no~savefill ( flags fillchar -- adjusted_flags  1/20/90 )
 swap -256 and or ;

: no~fillout ( flags -- adjusted_flags  1/20/90 )
 dup 255 and dup hold ascii $ = if bl no~savefill $2000 or then ;

: no~# (      value
               sign
           cursctrl
              flags
                 --
     adjusted_value
               sign
  adjusted_cursctrl
              flags  1/20/90 )
 3 pick if
   >r 2>r # 2r>    \ emit a digit, if there are any left
 else
   no~fillout >r   \ emit the fill character if no digits left
 then  no~bumpcursdigit r> ;
```

one represents the "tens" position, the power two the "hundreds" position, etc. The position also happens to be a convenient concept for digit incrementing and decrementing; see the words `ni~incrdigit` and `ni~decrdigit`. Another use for the position is in bounds-checking the addition and deletion of digits in the words `ni~adddigit` and `ni~deldigit`.

The number output routine plays a key role in this number input scheme. When a number is displayed (see the word `no~displayer`), both the value and the position are processed as the picture string is traversed from right to left, and in addition to returning the total length of the output string, the distance in characters from the right edge of the output string to the cursor position is returned as well. These two values are exactly what the number input routine must have to adjust the cursor back to its original position before attempting to display an updated value.

Three types of valid keystrokes are accepted by the number input routine: the first type is comprised of the digits and the sign characters (- toggles the sign of the current value, + takes the absolute value of the current value); the second type are editing keys (cursor movement and backspace); the third type are termination keystrokes (carriage return, space, and tab). The main input loop (`ni~doinput`) doesn't know or care what the first two types of keys are, it simply calls `ni~keyadj` to process the keystroke. If `ni~keyadj` recognizes the keystroke, it makes the appropriate adjustments to the state on the stack, and consumes the keystroke. If the keystroke is not recognized, it remains on the stack.

`ni~doinput` simply looks to see if the keystroke has been consumed: if so, it obtains another key and loops. If not, the keystroke is examined to see if it is a termination key, in which case `ni~doinput` exits. All other keystrokes are processed by notifying the user of their invalidity by beeping. Note that some keystrokes are valid (and therefore consumed) in some states but not in others.

Notice that because of the asymmetry of positive and negative values in two's complement arithmetic, it is possible to enter a negative number that is too big to negate, thus invalidating the + and - keystrokes until the number is reduced in magnitude.

These routines correctly handle this situation, but if you think it is too confusing for the user, it can be avoided either by reducing the maximum magnitude of negative numbers (`maxnegative`) by one, or by only using picture strings that are constrained to fewer than the maximum number of digits possible. Another curious state that can be avoided by using fewer than the maximum number of digits, or by changing the specified maximums, is the limited range of the leftmost digit due to the binary representation of non-binary numbers.

Comparison to Mike's `numin`

The particular points in Mike's code that were insufficient for my applications are described here, with some comments as to how mine differs in each respect, and the tradeoffs.

1. Use of a variable to maintain the context for `SIGNED` versus `UNSIGNED`. Use of a variable to maintain the number of expected digits. Use of a variable to maintain the number of current digits. Use of a variable to maintain the current picture string pointer.

Although all the state variables could become user variables, permitting re-entrancy in a multitasking environment, the number of user variables is limited on most implementations, so I chose to keep more information on the stack. Elimination of state variables also avoids the problem of making sure the state variables are all set to the proper values before each call to `numin`, which could otherwise be an easy source of program bugs. To avoid coding all the parameters to every call, additional words can be provided that supply constant values for some of the parameters: the need for such words and the values of the parameters they supply can be determined after using `numin` when starting to code a program—it will soon become obvious what sets of parameters change and which are constant for a given program or subprogram

2. `MAXDIGITS` is not a sufficient technique for preventing overflow in either the signed or unsigned case. I keep track of the number of digits in the picture and of the machine word-size limits, and restrict the user accordingly.

3. No support is provided for intranumber digit editing. This is the cause of

```
: no~out2 ( cursctrl
              flags
              char1
              char2
                 --
  adjusted_cursctrl
              flags  1/20/90 )
 hold hold >r no~bumpcurs no~bumpcurs r> ;

\ Picture codes 1 and 2 require an adjustment after emitting the last digit
\ to compensate for the adjustable spacing they use.
: no~fixslide ( cursctrl flags -- adjusted_cursctrl flags  1/20/90 )
 over 65535 and 255 > if swap 255 - swap then ;

: no~1out (    value
                sign
            cursctrl
               flags
                  --
      adjusted_value
                sign
   adjusted_cursctrl
               flags  1/20/90 )
 begin 3 pick while no~# repeat no~fixslide ;

: no~2out (    value
                sign
            cursctrl
               flags
                  --
      adjusted_value
                sign
   adjusted_cursctrl
               flags  1/20/90 )

 begin 3 pick while
   over 16 >> 3 /mod if
     0= if
       ascii , hold >r no~bumpcurs 65535 and r>
     then
   else
     drop
   then
   no~#
 repeat no~fixslide ;

: no~6out (   value
               sign
           cursctrl
              flags
                 --
     adjusted_value
               sign
  adjusted_cursctrl
              flags  1/20/90 )
 bl no~savefill 3 pick if
   no~#
 else
   dup $1000 and if
     bl
   else
     $1000 or over2 2 and if ascii - else bl then
   then
   hold >r no~bumpcurs r>
 then ;

\ rules for pictures: All characters not having special meaning are used
\ verbatim.  You can, of course, recode this to behave in exactly the way
```

```
\ you want your pictures to behave, but these give examples of possibilities.

\ Leading $ is verbatim, subsequent $ used as floating $.
\ 7 is used as * filling, 8 used as _ filling, 9 used as space filling,
\ 0 used as 0 filling.
\ 1 is used as an expandable digit field, 2 is an expandable digit field with
\   appropriately embedded commas.
\ , used as comma if there are preceding significant digits, as the filling
\   character if there was a filling character to the right of it, and no
\   preceding significant digits, and as a comma if no filling character to
\   its right. Also resets the digit position counter for implicit commas.
\ . is verbatim, but resets the digit position counter for implicit commas.

\ for signed numbers: + is a - for negative numbers, + for non-negative ones.
\ - is a - for negative numbers, space for non-negative ones.
\ ( and ) are themselves for negative numbers, spaces for non-negative ones.
\ 3 is CR for negative numbers, 2 spaces for non-negative ones.
\ 4 is DB for negative numbers, CR for non-negative ones.
\ 5 is CR for negative numbers, DB for non-negative ones.
\ Leading 6 is a sign position, subsequent 6 used as floating sign.

: no~dispchar ( value
                  sign
              cursctrl
                 flags
               fmtchar
                    --
        adjusted_value
                  sign
```

(Screens continued on page 16.)

my code being significantly larger than Mike's, but significantly eases the task of editing long numbers.

4. Limited picture codes. These could easily be added to Mike's implementation, but after I did mine, I felt it was more useful to add them to mine.

This code is placed in the public domain. You are welcome to examine it, retype it, modify it, give it away, or sell it. This code is fully tested and deemed to be working as submitted for publication; however, it carries no guarantee or warranty of fitness for any purpose.

Glenn Linderman has been using Forth for six years, mostly on a large program for music transcription. That program has been used to produce four books of sacred music with lyrics in four languages, and dozens of pieces of sheet music.

EXTENSIBLE OPTIMIZING COMPILER

ANDREW SCOTT - EDMONTON, ALBERTA, CANADA

Forth implementations on some processors suffer speed limitations. One solution to this problem has involved implementing a subroutine-threaded system to eliminate inner interpreter overhead. Short primitives are usually compiled as in-line code. This paper describes a method by which Forth can be compiled to produce more optimized code by combining sequences of Forth words into equivalent native instructions. The optimizer described does not require Forth primitives to be "smart" words, nor does it require extensive changes to the normal Forth outer interpreter. Also, the optimizer is extensible, permitting new optimization rules to be added at compile time.

Introduction

One criticism that has sometimes been unfairly applied to Forth is that it is not very fast. Granted, compared to code that an optimizing C compiler produces, the same algorithm expressed in indirect-threaded Forth on the same processor will execute more slowly. The mistake in making this comparison, however, is that apples are being compared to oranges. It would be more appropriate, for example, to compare indirect-threaded Forth to interpreted BASIC.

A common solution to the perceived speed problem has been to implement Forth in the native processor's machine language instead of using an inner interpreter. These subroutine-threaded Forths typically run twice as fast as indirect-threaded systems on the same CPU. Most of the advantages of Forth are still available, but tools such as decompilers and code-altering words are difficult to implement.

When I started work creating a subroutine-threaded Forth system, it became apparent that many optimizations could be made to further increase the execution speed of the compiled code. Many short primitive words, such as `DUP`, `+`, or `EXIT`, could be "in-lined" to avoid the overhead of subroutine calls and returns. This technique is very common in currently available subroutine-threaded Forths. One technique that I do not believe is as common, the subject of this paper, is the combining of sequences of Forth words into in-line code. For example, the common sequence `DUP >R` can be combined into one instruction on the 68000 CPU.

In this paper, I will discuss the extensible optimizer I created for use with a subroutine-threaded Forth system. The target CPU is a 68000, but most of the concepts I describe can be applied to any processor.

Writing an optimizing Forth compiler needn't require great effort.

Sequence Optimizations

The Forth virtual machine is a model of elegance and simplicity. The stack-based architecture is RISC-like in that it is not necessary to provide a dozen addressing modes for each instruction. CPU designers have realized this and have produced very powerful processors in a fraction of the chip space that conventional CPUs required. Alas, it is not always possible to use a Forth chip. If you must use a conventional CPU, try to use as many of the processor's features as you can. If a dozen addressing modes are available, use them.

It is very common for certain sequences of Forth words to appear together. For example, stack operations such as `DUP >R` and `SWAP DROP`, math operations such as `LIT +`, and memory operations such as `VAR @` and `VAR !` are some of the idioms a Forth programmer uses regularly. In fact, in some Forths these primitives have been combined to form other "primitives" in an effort to improve efficiency. For example, `NIP` is an alias for `SWAP DROP`, and `1+` and `2*` are used often for common math operations. The problem with this approach is that the kernel becomes littered with words that don't provide any additional functionality and make it more difficult for a newcomer to learn the language. I believe in the Forth philosophy that says, "Smaller is better."

A more general approach is to let the compiler worry about how to optimize these sequences. A set of rules describes how each sequence should be translated into native machine code. The compiler uses these rules to determine if an optimization can take place and how it should be done.

Compiler Operation

The traditional Forth compiler simply lays down an address or a token when a word is compiled. To optimize a sequence of words, it would be necessary to either delay the code generation until the sequence is complete or look back to what had been compiled previously. I chose the former approach.

When a word is compiled, its address is remembered in a list. If the word ends a sequence that can be optimized, the list is flushed and the optimized code is laid down. If a sequence ends prematurely, any partial optimizations that can be made from the list are compiled, and the remainder of the list is "recompiled" into a new list. If no partial optimizations can be made, the first address on the list is compiled with a default rule and the rest of the list is recompiled.

This algorithm is illustrated in Figure One. In this example, the fictional Forth words `A`, `B`, `C`, `X`, and `Y` are used. The sequences `A B C` and `X Y` can be optimized, and A can also be in-lined individually. The code to be compiled is `A B X Y`.

Optimization Rules

In the spirit of Forth, the optimizer I developed is extensible. Additional rules can be added to the rules database at compile time (run time, from the compiler's point of view). Each rule specifies the sequence of Forth words that can be condensed, and the Forth code that should execute when this sequence is encountered. Typically, this involves invoking the assembler to lay down the in-line code.

For example, some rules are given in Figure Two-a to translate common sequences of Forth words to 68000 code. In those examples, the word `SEQ:` is used to mark the start of each sequence. It looks ahead, "ticking" the following words until `IS:` is found. Code between `IS:` and `;` is compiled, and the address of this code and the addresses of each word in the sequence are added to the rules database. (The words `LIT` and `L>` will be explained in the next section.)

The data structure built when rules are compiled is shown in Figure Three. (In Figure Three, assume that the rules given in Figure Two-b have been defined.) The structure is shaped like a set of trees. Each tree's root is the common first word to a set of sequences. At each node of the tree is stored a pointer to the code that will be executed if the sequence ends at that node. A null pointer indicates that the sequence cannot be optimized at this point.

The link between the top two nodes indicates the list of nodes searched when a word is compiled onto an empty sequence list. Only rules beginning with `A` or `X` have been defined. If either of these words is compiled, the search begins with the node list underneath the top-level node when the next word is compiled. If a word is compiled that does not exist in the node list currently pointed to, the backtracking algorithm occurs as described above.

Note that nodes are shared by more than one rule. This makes it possible to discover and compile partial optimizations.

Optimizing Literals

Many of the optimization rules involve

Figure One. Optimization process using fictional Forth words and sequences described in the text.

Sequence list	Description
A	`A` is compiled. No code is generated at this point, as the sequence `A B C` could yet occur. An entry for `A` is put in the sequence list.
A B	`B` is compiled and remembered in the list.
A B X	`X` is compiled. The sequence `A B C` did not occur, so the list is reduced. A partial optimization can be made: `A` is in-lined.
B	`B` is recompiled, but it does not start a sequence, so it is compiled using the default compilation rule.
X	`X` is recompiled. It starts the sequence `X Y`, so it remains in the list.
X Y	`Y` is compiled. It ends the sequence `X Y`, so the optimized code is laid down and the list is flushed.

Figure Two-a. Translating common Forth sequences into 68000 code.

```
SEQ:  SWAP DROP      IS:    sp@+ sp@ mov ;
SEQ:  DUP >R         IS:    sp@ rp@- mov ;
SEQ:  LIT @          IS:    L> :1 sp@- mov ;
SEQ:  LIT +          IS:    L> #1 sp@ add ;
```

Figure Two-b. Optimizing the fictional Forth words used in Figure Three.

```
SEQ:  A B C          IS:    ( code to optimize A B C ) ;
SEQ:  A              IS:    ( code to in-line A ) ;
SEQ:  X Y            IS:    ( code to optimize X Y ) ;
SEQ:  X Y Z          IS:    ( code to optimize X Y Z ) ;
SEQ:  X A            IS:    ( code to optimize X A ) ;
```

Figure Three. The data structure built when rules are compiled.

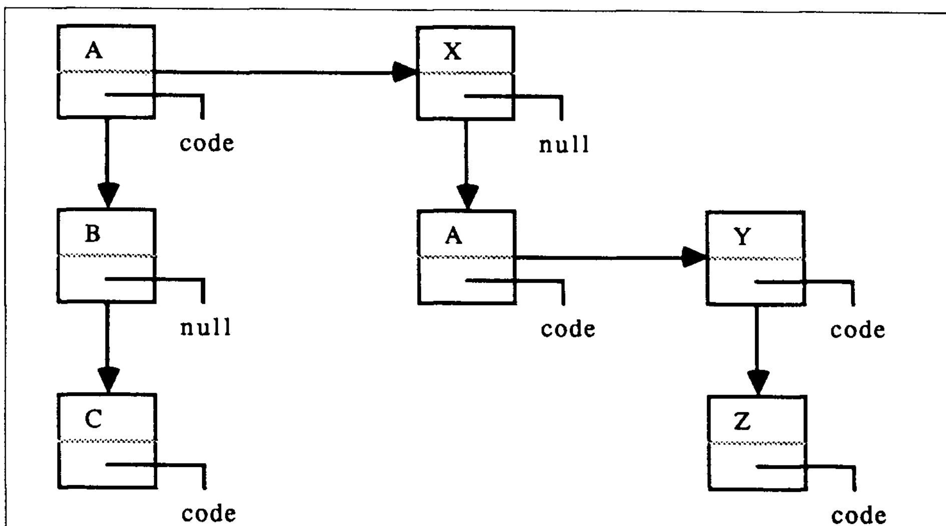

Figure Four. Common optimizations from the author's rules file.

Stack optimizations

```
SEQ:   SWAP DROP        IS:   sp@+ sp@ mov ;
SEQ:   DROP LIT         IS:   L> #1 sp@ mov ;
SEQ:   DUP >R           IS:   sp@ rp@- mov ;
SEQ:   R> DROP          IS:   4 #1 rp addq ;
```

Fetch/store optimizations

```
SEQ:   LIT @            IS:   L> :1 sp@- mov ;
SEQ:   LIT +            IS:   L> #1 sp@ add ;
SEQ:   LIT LIT !        IS:   L> L> ?DUP IF
                                #1 :1 mov
                              ELSE
                                :1 clr
                              THEN ;
SEQ:   LIT @ LIT !      IS:   L> L> :1 :1 mov ;
SEQ:   DUP LIT !        IS:   sp@ L> :1 mov ;
```

Math optimizations

```
SEQ:   LIT +            IS:   L> #1 sp@ add ;
SEQ:   LIT -            IS:   L> #1 sp@ sub ;
SEQ:   LIT OR           IS:   L> #1 sp@ or ;
SEQ:   LIT AND          IS:   L> #1 sp@ and ;
SEQ:   LIT XOR          IS:   L> #1 sp@ eor ;
```

Branch optimizations

```
SEQ:   = ?BRANCH        IS:   sp@+ sp@+ cmpm bne ;
SEQ:   <> ?BRANCH       IS:   sp@+ sp@+ cmpm beq ;
SEQ:   < ?BRANCH        IS:   sp@+ sp@+ cmpm bge ;
SEQ:   > ?BRANCH        IS:   sp@+ sp@+ cmpm ble ;
SEQ:   <= ?BRANCH       IS:   sp@+ sp@+ cmpm bgt ;
SEQ:   >= ?BRANCH       IS:   sp@+ sp@+ cmpm blt ;
SEQ:   LIT = ?BRANCH    IS:   L> #1 sp@+ cmpm bne ;
SEQ:   LIT <> ?BRANCH   IS:   L> #1 sp@+ cmpm beq ;
SEQ:   LIT < ?BRANCH    IS:   L> #1 sp@+ cmpm bge ;
SEQ:   LIT > ?BRANCH    IS:   L> #1 sp@+ cmpm ble ;
SEQ:   LIT <= ?BRANCH   IS:   L> #1 sp@+ cmpm bgt ;
SEQ:   LIT >= ?BRANCH   IS:   L> #1 sp@+ cmpm blt ;
```

(Number Input screens continued from page 13.)

```
   adjusted_cursctrl
           flags  1/20/90 )
\ flags is used to hold the current fill character, and high bits are used
\ for other flags.

case ascii 0 = of ascii 0 no~savefill no~# endcase
case ascii 1 = of no~1out endcase
case ascii 2 = of no~2out endcase
case ascii 7 = of ascii * no~savefill no~# endcase
case ascii 8 = of ascii _ no~savefill no~# endcase
case ascii 9 = of bl no~savefill no~# endcase
case ascii $ = of dup $2000 and 0= if ascii $ no~savefill then no~# end-
case
case ascii , = of 3 pick if ascii , hold else no~fillout then
   >r no~bumpcurs 65535 and r> endcase
case ascii . = if hold >r no~bumpcurs 65535 and r> endcase

3 pick 3 and 0= if \ for unsigned numbers, all others are verbatim
   hold >r no~bumpcurs r> endcase
```

"folding" a number or address into an instruction as an absolute address of an immediate value. In Forth, there are three ways of describing a value: as a `CONSTANT`, as the address of a `VARIABLE` (words defined with `CREATE` fall into this category also), or as a value compiled in-line with `LITERAL`. There really should be no distinction between these, as they all push a number to the stack when executed. In my Forth, `CONSTANT` and `VARIABLE` simply use the run-time code of `LITERAL`, `LIT`, to push the number or address to the stack.

Rules that do use a literal value need this value when the rule's code is executed. The literal value is remembered in another data structure, the literal stack, when it is compiled. When a word is added to the sequence list, the depth of the literal stack is also remembered in the list. When backtracking takes place for partial optimizations, the literal stack is restored to its state when the literal was initially compiled.

The executable code associated with rules involving literals uses the word `L>` to pop the top value from the literal stack. The value was put there initially by `LITERAL`, which is defined now as:

```
: LITERAL ( n -- )
  >L COMPILE LIT
  ;  IMMEDIATE
```

`L>` must be called as many times as `LIT` appears in the sequence.

Recursive In-lining

To remain compatible with regular Forth, it was necessary to add words such as `1+` and `NIP` to the system. These words could be optimized internally, but it would be better to in-line them in the words they were compiled in.

When short words like these are compiled, the optimizer is temporarily disabled. Instead, each internal word is compiled with the default rule. (In a subroutine-threaded Forth, the default rule is to compile a subroutine call to the word.) These words will never be called from compiled code, but only from the outer interpreter, so the lack of optimization does not really matter. A NOP instruction precedes the rest of the internal code.

The NOP is a signal to the optimizer that when this word is compiled, the component words inside it should be compiled instead. In effect, using subroutine calls only makes the word easily "decompilable" by the optimizer.

I use `::` to signify that one of these special words is to be compiled. Thus, the following definitions appear in the source code to my system:

```
:: NIP   ( a \ b -- b )
   SWAP DROP ;

:: 1+   ( n -- n+1 )
   1 + ;
...
```

When `NIP` is compiled inside another word, `SWAP` and `DROP` are compiled instead.

The benefit of this approach is that partial sequences can be coded in a `::` word for clarity, and then combined with the rest of the sequence in later code. For example, using the rules described for Figure Two, we could write:

```
:: AB
   A B ;
```

(this is the first part of `A B C`)

```
...
: AWORD
  DUP AB C DROP ;
```

When `AWORD` is compiled, `AB` will be expanded to compile `A` and `B` individually. The sequence `A B C` will complete when `C` is compiled next.

Previously I mentioned that variables, constants, and literals really do the same thing. In fact, `VARIABLE` compiles a `::` word with the literal value of the variable's address inside, and `CONSTANT` compiles a `::` word with the literal value of the constant inside. Thus, every constant and every variable defined will be optimized by one of the rules involving `LIT`.

Implementation Observations

The major difficulties I encountered in porting old code to the new optimizing Forth resulted from implementation-specific code. For example, some code changed the value of a `CONSTANT` by altering the parameter field of the word. This is a dubious practice to begin with, and failed miserably when the value of the constant was in-lined everywhere it was used. Another problem was the use of `]` and `[` to create a jump table:

```
case ascii - =
  of over2 2 and if ascii - else bl then hold >r no~bumpcurs r> endcase
case ascii + =
  of over2 2 and if ascii - else ascii + then hold >r no~bumpcurs r> endcase
case ascii ( =
  of over2 2 and if ascii ( else bl then hold >r no~bumpcurs r> endcase
case ascii ) =
  of over2 2 and if ascii ) else bl then hold >r no~bumpcurs r> endcase
case ascii 3 =
  of over2 2 and if ascii C ascii R else bl bl then no~out2 endcase
case ascii 4 =
  of over2 2 and if ascii D ascii B else ascii C ascii R then no~out2 endcase
case ascii 5 =
  of over2 2 and if ascii C ascii R else ascii D ascii B then no~out2 endcase
case ascii 6 = of no~6out endcase

 \ all others are always verbatim
 hold >r no~bumpcurs r> ;

: no~formatter ( value
              position
                  sign
          formatstring
                    --
            cursorbksp
                outptr
                outlen
            cursorctrl  1/20/90 )
 rot
 \ calculate the initial cursor positioning control
 0 swap begin dup while base @ / swap 1+ swap repeat drop 8 <<
 \ add flags for no~dispchar to use, setup loop indices
 swap 0 swap ( v s cc fl fs - ) <# count range swap 1- do
   i c@ ( v s cc fl char - ) no~dispchar \ process each character
 -1 +loop drop -rot drop #> over2 65535 and over 8 << + ;

: no~coverup (new_width old_cursorctrl -- 1/20/90 )
 8 >> swap - dup 0> if dup spaces for 8 emit next else drop then ;

: no~displayer ( value
              position
                  sign
          formatstring
            coverwidth
                    --
            cursorctrl  1/20/90 )
 >r no~formatter r> swap 2>r dup>r type 2r> no~coverup no~dispcurs r> ;

\ format numeric output via format strings
: numformat ( value
              sign
       formatstring
        number_base
                 --
            outptr
            outlen  1/20/90 )
 base @ >r 36 min base ! 0 -rot 0 no~formatter drop rot drop r> base ! ;

: numoutput ( value sign formatstring number_bas --  1/20/90 )
 numformat type ;

\ print numeric output via format strings
: numprint ( value sign formatstring number_base --  1/20/90 )
 numformat typep ;

: ni~istermkey ( key -- flag  1/20/90 )
 dup 13 = over 9 = or swap bl = or ;

: ni~outadj ( cursorctrl --  1/20/90 )
 dup 255 and dup 1 > if
   1- for bl emit next
 else
   drop
 then
 8 >> for 8 emit next ;
```

(Continued on next page.)

```
: ni~doinput ( initial_value
                    position
                     signed?
               output_width
               formatstring
                  first_key
                         --
                final_value
                   position
                       sign
               output_width
               formatstring
                   term_key  1/20/90 )
 swap >r swap >r
 begin
   ni~keyadj \ see if we can process this key
   ?dup if \ key not consumed
     dup ni~istermkey if ( fv p s tk - fs ow - ) 2r> rot exit then
     drop beep \ indicate error
   then
   3dup r@ ni~outadj
   r> r@ swap no~displayer >r
   key 127 and
 again ;

\ adjust for additional digit
: ni~adjdigit ( pos
           #digits
            $6flag
                --
      adjusted_pos
   adjusted_#digits
            $6flag  1/20/90 )
 2>r us>d base @ um/mod swapdrop r> 1+ r> ;

: ni~isnumeric ( morepos
                 #digits
                 intflag
                    char
                      --
        adjusted_morepos
        adjusted_#digits
        adjusted_intflag  1/13/90 )
 case ascii $ = of dup $40 and if ni~adjdigit else $40 or then endcase
 case ascii 6 =
   of dup $80 and if dup $20 and if ni~adjdigit else $20 or then then endcase
 case ascii 7 = ascii 8 =or ascii 9 =or ascii 0 =or
    of ni~adjdigit endcase
 case ascii 1 = ascii 2 =or of >r 2drop 0 dup r> endcase ( use machine limit )
 drop ;

\ count number of digits permitted in the formatstring, and if that would be
\ the limiting factor on number range, return the number of digits.  Oth-
erwise,
\ return 0 to use the maximum natural machine range as the limit.

: ni~countdigits ( sign formatstring -- number_digits|0  1/20/90 )
 >r 3 and dup>r ni~max 0 r> if $80 else 0 then ( indicate signed-ness )
 r> count range do i c@ ni~isnumeric loop
 ( adjust stack, noticing if the digit count exceeds the natural machine
limit,
   and if so, substituting 0 to indicate the natural machine limit for the
    digit count. )
 drop swap 0= if drop 0 then ;


\ obtain numeric input
\ flags bits are: bit 0 = 0, let user edit default value,
\                       = 1, user accepts default, or replaces with new
value

: numinput ( initial_value
                   signed?
              formatstring
                     flags
```

```
CREATE JUMPER
] ACTION0 ACTION1 ACTION2 [
... JUMPER + @ EXECUTE ...
```

A more portable version that the optimizer wouldn't touch is:

```
CREATE JUMPER
' ACTION0 ,
' ACTION1 ,
' ACTION2 ,
```

The other problems I encountered resulted from using a subroutine-threaded Forth. For example, as commonly defined, `COMPILE` cannot be written on these systems. The usual solution is to make `COMPILE` immediate.

What is really needed is a word analogous to `EXECUTE`. It would accept the ticked address of a word and compile it. I wrote a word called `(COMPILE)` that works this way. On most systems, `(COMPILE)` would simply comma the address into the dictionary. On my system, `(COMPILE)` invokes the optimizer. In Forth, we have some portable words for writing compiler words such as `<MARK` and `<RESOLVE`. We just need to complete the set.

To get a better idea of some of the common optimizations that can be done, a small portion of my rules file is shown in Figure Four.

Conclusions

Forth need not be a slow language. An optimizing Forth compiler can be written without requiring a great deal of effort. Granted, my optimizer still doesn't compare equally to an optimizing C compiler, but it's fairly close and it didn't require several man-years to write.

On the 68000 CPU, subroutine-threaded Forth executed about twice as fast as indirect-threaded Forth. With the optimizer added to the compiler, the code ran about three times as fast as indirect-threaded Forth. Thus, the optimizer added a 50% improvement to the execution speed of the programs I compiled for comparison purposes.

Benchmarks and timing tests should always be taken with a grain of salt, but coding a bubble sort algorithm in Forth, C, and 68000 assembly language yielded the following results (in seconds):

indirect-threaded Forth	97
subroutine-threaded Forth	48
subroutine-threaded Forth with optimizer	27
C	15
hand-tuned 68000 assembly language	07

The time to execute the C function reflects the register nature of the language and the speed of register-addressing modes on the 68000. Using registers to represent the top stack items in Forth would be an interesting experiment.

I have just begun to experiment with other optimization techniques. Compile-time arithmetic (e.g., VAR 4+) and other kinds of "expression folding" would be another effective addition to the Forth compiler. What makes this kind of experimentation easy is the extensibility and interactive nature of the Forth language.

Reprinted from the FORML *Proceedings 1989. Andrew Scott has been programming in Forth since he received his B.Sc. in Computer Engineering from the University of Alberta in 1986. He works for IDACOM Electronics Ltd., where he is using Forth to develop a language to describe components of ISDN protocols.*

ADVERTISERS INDEX


```
              number_base
                       --
              final_value
                 term_key  1/20/90 )
 base @ >r 36 min base ! 2>r 1 swap        ( i_v pos s? -  onb fl fs - )
 3 and dup r@ ni~countdigits 4* + \ adjust sign flag to contain max # digits
 3dup r@ 0 no~displayer 2r> key 127 and ( i_v pos s? ow fs fl key - onb - )
 dup ni~istermkey
 0= if
   \ not terminator key
   swap if 2>r 2>r >r drop 0 r> abs 2r> 2r> then \ default not used - set
to zero
   ni~doinput \ process the input
 else
   swapdrop
 then
 ( i_v pos s? ow fs key -  onb - )
 2>r 2>r drop dup r@ 2 and if negate swap then \ save result with correct
sign
 0 2r> ni~outadj r> 0 no~displayer drop ( i_v -  onb key - )
 2r> base ! ;
: test ( num sign pic zflag base --  1/20/90 )
 2 selcur \ turn text cursor on
 numinput
 0 selcur \ turn text cursor off
 ascii ! bkemit
 cr " Termination char:" count bktype h. " Final value:" count bktype
. ;
\ test leading zeros, 5 digit limit, 2 minus signs
: t1 ( --  1/20/90 )
 37 1 " -00000-" 0 10 test ;
\ test leading spaces, 6 digit limit
: t2 ( --  1/20/90 )
 798 1 " -999999-" 0 10 test ;
\ test all the different sign indicators, even the leading "-", and
\ comma suppression
: t3 ( --  1/20/90 )
 144 -1 " - + 3 4 5 (6,660.00)" 0 10 test ;
\ test leading "$"
: t4 ( --  1/20/90 )
 179 1 " $$$,$$0.00 4" 0 10 test ;
\ test leading "*"
: t5 ( --  1/20/90 )
 3 1 " $77,770.00-" 0 10 test ;
\ test leading " "
: t6 ( --  1/20/90 )
 43 1 " $88,880.00 5" 0 10 test ;

\ test variable width field
: t7 ( --  1/20/90 )
 77 1 " -$1" 0 10 test ;

\ test variable width field with embedded commas
: t8 ( --  1/20/90 )
 88 1 " -$2" 0 10 test ;

\ this case looks like money input
: t9 ( --  1/20/90 )
 99 0 " -$2.00" 0 10 test ;

\ this case tests variable width field in combination with other digit types
\ also tests unsigned numbers
: t10 ( --  1/20/90 )
 111 0 " 4 $290" 0 10 test ;

\ the digits don't have to be adjacent, and cursor still tracks correctly
: t11 ( --  1/20/90 )
 121 0 " 9 9 9 0 0 0" 0 10 test ;

\ test a base other than decimal
: t12 ( --  1/20/90 )
 131 1 " (2)" 1 36 test ;
```

Part One

FORST: A 68000 NATIVE-CODE FORTH

JOHN REDMOND - SYDNEY, AUSTRALIA.

This three-part series of articles will describe a 32-bit Forth based on the TOS operating system of the Atari ST. TOS (Tramiel Operating System) is pretty much a 68000 clone of MS-DOS. The calls have the same numbers and functions, but differ in that the parameters are passed on the stack rather than in registers. The directory structure is identical to that of MS-DOS, and data files are completely portable on 3.5" disks.

The ForST source code is in several function files, which are referenced by a single top-level load file. Depending on user requirements, some of the files can be removed, along with the corresponding header entries. In this way, the size of the ForST system can be controlled. Cultural aspects like vocabularies and a screen editor are not included in the system at present, but can be added later.

While the system has a number of non-standard characteristics, attention has been given to its compatibility with existing source code. With some small qualifications about the header structure, and state-smart variables and constants, it is close to 100% compatible.

I've tried to take note of (perceived) limitations of the traditional Forth disc I/O and to incorporate some of the better features of other languages and other operating environments; so I've incorporated the use of multiple files and redirectable buffered I/O. ForST carries out all its I/O by BIOS calls to the TOS firmware.

System Characteristics

a. Extended (32-bit) addressing
The 68000 CPU has the advantage of a 16 Mbyte flat address space, all of which can be accessed without the intricacies of segment registers. To take advantage of the addressing range, however, it is necessary to have full 32-bit addressing (24-bit, really). Code and data size are limited only by the available memory.

b. Position-independent code
TOS, and probably all future operating systems, makes up its own mind about where a program is to be loaded; so it is also necessary to have a Forth written in position-independent code.

c. 32-bit stack width
If the Forth is to handle 32-bit addresses, it is appropriate to have a 32-bit width for all stack entries. This has the advantage of giving integers a much more useful range, and of making them the same size as IEEE short reals.

It is important to have loops with the correct behavior.

d. Subroutine-threaded code
This provides greater execution speed, removes the distinction between code and high-level words, and allows code optimization.

e. Macro definitions with edge optimization
Each word available for in-line expansion has an associated code flag and length marker to direct expansion and optimization.

f. Separated headers, with selective saving and deletion
At present, the headers are in a simple, unlinked list from which they can be selectively retained or removed (regardless of code type).

g. Good execution speed
100 iterations of the Sieve in 59 seconds, or 165 seconds without macro expansion.

How Wide?

The problem of word size has been discussed in detail [Bra87]. I support the view that the natural, default fetch-and-store words (`@` and `!`) should universally apply to movements corresponding to the stack width. We are now nibbling at 32 bits, but what in the future? When we get to 64 bits (and we will), `@` and `!` should apply to 64-bit operations. Provided that we get into the habit of using the natural words for the default data size, code will always be portable upwards.

Of course, there will always be a need to access smaller units, such as with `C@` and `C!`, and now we need `W@` and `W!` to cope with specialized applications which demand the 16-bit width (such as the GEM interface). This has been my approach with ForST. Variables, constants, and addresses are all 32 bit, and we use `@` and `!` with them—as always in the past.

The need for fetching and storing doubles may all but disappear as the default size increases, but `2DUP` and its related stack words will certainly stay because of their use with pairs of values.

The Headers

ForST has been designed with separate, dispensible, headers. The headers of any Forth system allow the outer interpreter to locate code and to either compile or execute it. As a program becomes developed and low-level words become incorporated into definitions of more complex words, their headers become superfluous. They take up memory, they may conflict with access to words with identical names, and they slow down dictionary searches. What is needed is a means of selective retention of only those headers which need to be accessed later, and a means of disposing of the others. This concept has been discussed before [Joh87], but there were difficulties in implementing it in the F83 environment, especially with respect to `DOES>`.

The header structure has been chosen as optimal for copying from one part of memory to another. The traditional arrangement has the headers dispersed through the code and connected as a linked list. The linking is necessary because of the varying amount of code between headers. ForST, on the other hand, keeps its headers abutted together in their own buffer. `THERE` (*cf.* `HERE`) returns the address of the first free byte after the last of the headers.

There are four fields in a header (in order): name field, flag field, code field, and parameter field. The flag field holds the word length (less the final `RTS`) and the edge marker, and is used by the compiler to direct macro expansions. The latter two fields (four bytes each) have address offsets (see below), and the name field will be of varying length and padded, if necessary, to an even number of bytes. As usual, the first and last bytes have bit seven set to allow `TRAVERSE` to find its way.

Header Structure

nfa: length+$80, 'name', (+ pad byte?)
ffa: length (=bytes/2), macro flag
cfa: offset address of code
pfa: offset address of data or code

`(FIND)` works its way back from `THERE` by executing code equivalent to

```
13 - -1 TRAVERSE
```

until it finds a match or runs against a base address, rather than the zero pointer of other systems.

A consequence of the header structure is that there is no special relationship between headers. Any of them can be located, its length calculated from the name length, and copied to another part of memory. The main header list can then be truncated at any point and the reserved headers copied back to the new position given by `THERE`.

A minor result of the header structure is that searches are a little slower than with optimal linked headers with the link field before the name field (but it is still pretty fast), but selectively discarding headers compensates for this. A more important consequence is that there is presently no system of vocabularies. Whether or not this is a critical disadvantage depends on perspective. To date, mine has been on development of application code rather than a total environment, and for that the module approach is better. Moreover, once the system is up and going, the heads can be reorganized into any number of linked lists, or any other sort of data structure. Then, because `FIND` is `DEFER`ed, it can be redirected to search the new data structure.

Creating Modules and TOS-executable Applications

Disposable heads lead to the concept of a program module. After definition of a module, a small group of words will be chosen as `PUBLIC` by saving their headers, while the rest will implicitly be local and have their headers discarded. This is in the best tradition of information-hiding.

To implement this concept, ForST uses the dummy words `PROGRAM` and `MODULE`. As illustrated, as each module is completed, only some of its words remain accessible—and when the program is completed, only selected headers are kept.

```
: PROGRAM ;

: MODULE ;
VARIABLE FIRST
CONSTANT TRUE, etc.
: DEF1 ;
: DEF2 ;
...
: DEFN ;
FROM MODULE
KEEP TRUE
KEEP DEFN
PUBLIC
(only TRUE and DEFN are made public)

: MODULE ;
(list of definitions again)
FROM MODULE
KEEP ...
KEEP ... (etc.)
PUBLIC

: MODULE ;
(this might be the final module)
(several definitions again)
FROM MODULE
KEEP ...
PUBLIC (optional at end)

FROM PROGRAM
KEEP ...
PUBLIC
(this might be a single word from the whole program)
```

It follows that, even if a very large program is compiled into the basic ForST, it will have almost no effect on the speed of dictionary searches. Furthermore, almost all Forth system words can be made local (and therefore inaccessible) by the global `KEEP`:

```
FROM START
KEEP <application_name>
PUBLIC
```

From this point, only the application and the two words `SAVE` and `SYSTEM` are available to the user. `SAVE` allows us to save to the disk a standalone application, and `SYSTEM` allows us to get back to the GEM benchtop:

```
SAVE  A:\PATH\APPNAME.TOS
SYSTEM
```

There are some finer points to specifying how much work and header space the application will need, and whether an auto-exec is required on reloading but, otherwise, it is just that simple to generate a machine-code application.

How to Optimize Optimally

When primitive words such as `DUP` and `+` are used intensively in code, much of the execution time is taken up with pushes to and pops from the parameter stack. Consider the high-level definition of `2*`:

```
: 2* DUP + ;
```

Ignoring the overhead of subroutine calls and returns, the active code will be something like the steps below. If `DUP` and `+` are expanded as macro primitives, steps three and four are brought into sequence. It now becomes clear that they are very inefficient. They are expensive in terms of clock cycles, and it is the task of the edge optimizer to recognize and remove them.

(call `DUP`)
1. push the top stack value onto the stack (return and call +),
2. pop the (same) value to a register,
3. add the (now) top stack value to it,
4. push the (result) value to the stack.

(return)

To make the example specific to the 68000

Figure One-a.

```
(DUP)
1. MOVE.L     (A6),-(A6)        (20/2)    (+)
2. MOVE.L     (A6)+,D0          (12/2)
3. ADD.L      (A6)+,D0          (14/2)
4. MOVE.L     D0,-(A6)          (12/2)
Total: 58 cycles/8 bytes
```

Figure One-b.

```
(DUP)
1. MOVE.L     (A6),D0           (12/2)    (+)
2. ADD.L      (A6)+,D0          (14/2)
3. MOVE.L     D0,-(A6)          (12/2)
Total: 38 cycles/6 bytes
```

Figure One-c.

```
(I)
1.   MOVE.L   D6,D0                    (4/2)
2.   ADD.L    D7,D0                    (6/2)
3.   MOVE.L   D0,-(A6)                 (12/2)   (DUP)
4.   MOVE.L   (A6),-(A6)   Remove      (20/2)   (+)
5.   MOVE.L   (A6)+,D0     Remove      (12/2)
5a.  MOVE.L   (A6),D0      New         (12/2)
6.   ADD.L    (A6)+,D0                 (14/2)
7.   MOVE.L   D0,-(A6)     Remove      (12/2)   (+)
8.   MOVE.L   (A6)+,D0     Remove      (12/2)
9.   ADD.L    (A6)+,D0                 (14/2)
10.  MOVE.L   D0,-(A6)                 (12/2)
Total: 62(106) cycles/12(18) bytes
Subroutines: 254 cycles/12 bytes
```

[Cha87], the steps correspond to those in Figure One-a. If the steps are part of subroutine calls, a minimum of a further 68 clock cycles would be required (126 cycles total), but the size would still be eight bytes.

One of the real advantages of the 68000 is the orderly set of addressing modes for moves. There are no specific push or pop instructions, although the hardware stack is addressed by register A7. ForST uses A6 as the data-stack pointer. Therefore, a push of register D0 to the data stack will be coded:

```
MOVE.L        D0,-(A6)
```

and a pop from the return stack is:

```
MOVE.L        (A7)+,D0
```

It is nevertheless more concise and descriptive to refer to the processes of pushing and popping, so I will use this terminology.

Moves to and from memory, and the eight data registers and eight address registers, have very similar opcodes. This simplifies the operation of an edge optimizer, which balances and redirects moves. Optimization of the preceding code is shown in Figure One-b. The `DUP` memory-to-memory move and the following pop are converted into the more efficient memory-to-register move. The speed optimization is significant, even with this short fragment of code, and the size is now smaller than for two subroutine calls. Furthermore, the code has a trailing push, which provides scope for further optimization when incorporated into a larger definition.

To reinforce this point, consider the use of our new `2*` as a macro in the following code segment:

```
... I 2* + ...
```

In the expanded code shown in Figure One-c, the edges removed or modified by the optimizer are marked to show that incorporation of `2*` as a macro into a larger definition allows exploitation of its edge.

The outcome is code which is much faster than a simple macro expansion, and which still has an edge for further optimization. In non-trivial code, expansion/optimization will typically give a speed improvement by a factor of 2.5–3.0 over subroutine calls, with little change in code size.

Control of Macro Expansion

During compilation of many non-immediate words, we may have the option of compiling a call to the code of the word, or of doing a direct copy of its code into the word being compiled, provided it is not too long. We control this option by setting the macro compilation mode to false (with `CALLS`) or true (with `MACROS`). `CALLS` and `MACROS` are themselves immediate words and can be used for dynamic control of the compilation process within word definitions.

Not all words are appropriate for macro expansion, however. Any code which has a branch to outer code will not be suitable and will have a false macro code flag. In this event, the compiler will test the flag and proceed to compile a call. If the macro edge marker is non-zero, the macro compilation mode is true, and the word length is not longer than the preselected maximum in the variable `LONGEST`, the code will be expanded and the edge marker will be used to direct the optimization process.

Implementing Edge Optimization

There are two groups of edge markers: the push group and the negative group (-1 and -2). Before expansion of a word with a push marker, a flag is tested to determine whether an expansion has just taken place. If so, the last two bytes of code (the end of the previous expansion) are tested. If they are identical to the marker, they are removed *and the first two* bytes of the present word are skipped. The result is four bytes less code and 24 clock cycles less execution time. If the two bytes do not match the marker, they will be tested more generally for a push opcode. If this proves true, it indicates that the instruction can be altered to give more efficient code. As an example:

```
PUSH      D1
POP       D0
Total: 4 bytes/24 cycles
```

converts to:

```
MOVE.L    D1,D0
Total: 2 bytes/6 cycles
```

This is still a significant level of optimization, and it illustrates the advantage of starting a macro primitive with a pop and ending it with a push.

The simpler case of a negative edge

marker directs expansion of in-line code without optimization at the leading edge. A value of -1 is used by some system words, such as OVER which does not start with a POP, and -2 is generated in some special cases of user definitions. A user definition which consists solely of macro expansions (and which may include IF, BEGIN, etc.) will take on the macro flag of the first expanded word in its definition. Inclusion of one or more calls in the definition will zero the edge marker.

Loops and Optimization

Programmers who demand speed at any cost attach a great deal of importance to fast loops. It is very important, however, to have loops with the correct sort of behavior. The Forth-83 Standard loops have been problematic when the initial indices are identical, as they lead to a number circle excursion rather than a quick termination. This might be acceptable for 16-bit loop indices, but 32 bits are another matter. It might take a lifetime to recover from an uncontrolled loop! To cope with this, ForST incorporates a test on entry to the loop. If the indices are equal, the loop is skipped entirely. The behavior is safe and reasonable (*cf.* ?DO of F83) but differs from the Forth-79 Standard, which would have allowed one passage through the loop. This is my solution to a potentially dangerous problem, but I am aware that better approaches may be in the wind.

Because the 68000 has a good complement of registers, it is possible to assign specific tasks to some of them. Code for DO ... LOOP and +LOOP is very efficient because registers D6 and D7 are reserved for the indices. To allow nesting of loops, registers D6 and D7 are saved to a special loop stack by (DO) before entering the loop and are restored after leaving it. This means, of course, that >R and R> will have no effect on the progress of the loop, and that R@ and I will give different results. The 32-bit value in register D6 is incremented at the end of each iteration and the loop repeated if the overflow bit is clear.

The loop code takes only six bytes and 16 clock cycles, which is not much time at eight MHz. There is a single decrement-and-branch instruction (DBRA) for the 68000, which would take only four bytes and ten cycles, but it uses only a 16-bit value in the register (one of the handful of

Figure Two. Forth source code for the Sieve.

```
DECIMAL
MACROS
8190 CONSTANT SIZE
CREATE FLAGS SIZE ALLOT

: DO-PRIME  FLAGS SIZE 1 FILL
   0 SIZE 0
   DO FLAGS I + C@
      IF I DUP + 3 +  DUP I +
         BEGIN DUP SIZE <
         WHILE 0 OVER FLAGS + C!
                OVER + REPEAT
              DROP DROP 1+
      THEN
   LOOP
   ( . ." primes " adds 0.05 sec per loop)
   DROP

: PRIMES  0 DO DO-PRIME LOOP ;
```

Figure Three. Annotated ForST object code for DO-PRIME.

```
_do-prime:      lea         flags,a0
                push        a0          ;move.l     a0,-(a6)
                move.l      #size,d0
                push        d0
                bsr         _fill
;
                moveq       #0,d0
                push        d0
                move.l      #size,d0
                push        d0
                moveq       #0,d0
                push        d0
                bsr         bloop       ;install loop indices
                beq         lpescape    ;quit if indices equal
;
lpstart:        lea         flags,a0
                push        a0
                move.l      d6,d0
                add.l       d7,d0       ;I
                add.l       (a6)+,d0    ;+
                move.l      d0,a0
                moveq       #0,d0       ;clear 32 bits
                move.b      (a0),d0     ;fetch byte
                beq         notset
;
                move.l      a6,d0
                add.l       a7,d0            ;I
                push        d0
                move.l      (a6),d0          ;DUP
                add.l       (a6)+,d0         ;+
                push        d0
                moveq.l     #3,d0
                add.l       (a6)+,d0         ;3 +
                push        d0
                move.l      (a6),-(a6)       ;DUP
                move.l      d6,d0
                add.l       d7,d0            ;I
                add.l       (a6)+,d0         ;+
                push        d0
```

(Figure continues on next page.)

mistakes the 68000 designers made!).

LEAVE is implemented using another dedicated stack. The familiar linked list approach to marking addresses for forward branches is messy to use, as all addresses are offsets and all branches are relative.

Branch Optimization

The most basic elements of structured control are the words IF, ELSE, and THEN. For an indirect-threaded code interpreter, they are implemented by incorporating BRANCH and ?BRANCH into the code. These are the Forth equivalents of BRA and BEQ microprocessor instructions and, inevitably, are much slower. BRANCH, ?BRANCH, and the associated compilation words are available in ForST but, when invoked, they carry out direct compilation of BRA and BEQ instructions. As a result, the branching code is as efficient as that from any compiler. Structure checks are made during compilation of the structures by using marker values on the stack.

Similarly, BEGIN ... UNTIL and BEGIN ... WHILE ... REPEAT compile directly to patterns of single CPU instructions. Because all branching is by means of machine instructions and all destinations are within the code of a single word, the code macro flag of the definition is not affected and the word can, if it is shorter than the maximum length in the variable LONGEST, be expanded as in-line code when it is used in higher-level definitions.

Benchmarking ForST

An illustration of real code will demonstrate much of the previous discussion. The Sieve of Eratosthenes is a common and rather overworked compiler benchmark. In the past, it has been used to demonstrate that Forth is very inefficient by comparison to C. The Forth source code (often seen in these pages) is shown in Figure Two and the annotated ForST object code for DO-PRIME in Figure Three. This has resulted from compilation with MACROS and is, in fact, slightly *smaller* than the code with CALLS.

If this example does nothing else, it demonstrates the compactness and legibility of Forth compared to assembly! A close examination in the light of code for the primitives will suggest where edge optimization has taken place, and where code expansion has taken place without optimization. The most time-critical code is in the

```
begin:      move.l      (a6),-(a6)
            move.l      #size,d0
            cmp.l       (a6)+,d0
            sgt         d0
            ext.b       d0
            ext.w       d0            ;32-bit flag
while:      beq         wend
            moveq       #0,d0
            push        d0            ;0
            move.l      4(a6),d0
            push        d0            ;OVER
            lea         flags,a0
            move.l      a0,d0
            add.l       (a6)+,d0      ;+
            move.l      d0,a0
            pop         d0            ;0
            move.b      d0,(a0)       ;C!
            move.l      4(a6),d0      ;OVER
            add.l       (a6)+,d0      ;+
            push        d0
            bra         begin
;
wend:       pop         d0
            pop         d0            ;two DROPs
            pop         d0
            addq.l      #1,d0
            push        d0            ;increase count
notset:     addq.l      #1,d6
            bvc         lpstart       ;loop if no overflow
;
lpesc:      bsr         bunloop       ;get outer loop indices
            bsr         _dot
            bsr         _dotq         ;print string
                        rts
```

BEGIN ... WHILE ... REPEAT construct, where most of the action takes place and where every instruction really counts (it is traversed 14996 times to find the 1899 primes). The unconnected DUP at the start, the sign extension of the Boolean flag to 32 bits, the lack of a compare-immediate instruction, and the absence of direct-address arithmetic for pointers have added to execution time. More generally, the policy decision of using a 32-bit stack for all values has added 50% to all moves. I still believe the approach is correct, as it eliminates an important source of error at the source code level.

Nevertheless, 100 iterations of the sieve take only 59 seconds. Code from my Laser C compiler takes 41.7 seconds with 16-bit integers and 51.7 seconds when they are expanded to 32 bits. The code from ForST and the C compiler must be judged fairly equivalent in speed and quality. Intel enthusiasts will point to faster times for 80286 systems, but they should remember that they are using a narrow stack and the small memory model. They should recompile for long integers and the huge model, and compare again. Further to this point, the 68000 is near the bottom of the MC680X0 range and the ForST code will execute much, much faster on 68030 systems. The thought that code I am writing now will be usable in the computers of ten years hence is very appealing.

Finally, to confuse comparisons further, consider the following (useless) code, which is designed to test both the speed of looping and the time for subroutine calls.

```
DECIMAL CALLS

: NOOP ;
: RAWSPEED
  5000000 0 DO
  NOOP LOOP ;
```

(36 seconds)

Compiled with CALLS (to force a subroutine call to NOOP), it runs in 36 seconds. The corresponding code from Laser C takes 110 seconds, indicating a little of the very

(Continued on page 40.)

DICTIONARY STRUCTURES & FORTH

WU QIAN - BEIJING, CHINA

Forth has its own unique structure. One dictionary, a pair of stacks, and a dictionary-management system (DMS) comprise Forth. The dictionary includes all the functions of a Forth system so, along with the DMS, it forms Forth's framework.

Dictionary Structure System

The design of an operating system—even a single-user, single-task system—is a very complex task. Anyone who has designed a Forth system will deeply appreciate that it is much the same—though simpler, smaller, more flexible, and more effective than a generalized OS—but requires a shorter lead time to develop. Why?

The early designers of operating systems adopted the modular method, resulting in a modular-structure OS. The method is:

- divide a large-scale system design into small, independent, functional modules and stipulate the interfaces between the modules;
- implement the modules individually; then
- link the modules, according to their various interfaces, to form a complete system.

The main advantages of this modular approach are faster design, improved flexibility, and efficiency. Unix is a typical example of the method. The disadvantages include the difficulty of accurately parsing functions into modules and defining their interfaces in advance, and a lack of orderly call sequences reduces the relative independence of modules.

In order to improve on this method, a system of ordered calls is used. We may begin with a bare machine *A* which has no software; install a layer of software to expand the functions to form the virtual machine *A1*, which has better performance; then add another virtual machine *A2*, etc., until we arrive at the desired virtual machine *An*. The operating system designed with this method is divided into layers, each composed of modules, which have one-way relations. This is called a *hierarchical operating system*. Its design corrects the random call to the ordered call, which enhances design correctness and shortens development time. The lower layer can be designed and tested before the upper [or outer] layer, reducing the complexity of each programming task. Moreover, mistakes can be made later in the upper layer and subsequently corrected without breaking the lower layer's code, making maintenance much easier.

The development of Forth has just begun.

An operating system is the set of programs that controls and manages computer hardware and software resources, organizes the work flow, and generally makes things more convenient for the user. In fact, to design an operating system is to decide what method and structure should be used to organize this set of programs effectively. Resource management should also be handled by the operating system; from the user's point of view, this manages the programs that manage the hardware and non-system software resources. So a good operating system design should include not only the functions it is to realize but also, and more important, the management-control mechanics of the OS that make up its frame.

Neither modular nor hierarchical systems regard system programs as a resource to be managed in a fixed way. Modular systems don't manage the system programs at all—modules connect haphazardly, related merely as the caller and the called, resulting in unreliability and in long development and maintenance cycles. Hierarchical systems are more thoughtful, dividing the system into layers and managing the modules within those layers, but this reduces efficiency because of the forced division of similar resources and the relative isolation of the layers.

How can one manage system programs wisely in terms of resources? System programs are objects to be managed, and we can use tables to express their various resources. Their attributes differ because their functions differ, but each is an independent module. We can use one table to describe each module's name, hierarchical relationships, and other attributes. Modules and module tables together form the resource base of the system programs—a dictionary, in Forth terminology. The tools used to work with the dictionary comprise the DMS.

The dictionary is composed of modules and module tables, linked by a fixed data structure. Modules follow a strict calling order: modules in higher layers can only call modules in lower layers, a hierarchical relation included in the module tables. The DMS manages and controls the dictionary by providing operations to find a module, add a new module, remove a module, etc.

This dictionary structure system (DSS) differs from the simpler modular and hierarchical systems. The design of a DSS should first determine the dictionary structure and DMS. The designer can then use hierarchical methods to realize system functions that take full advantage of the system frame.

Forth is a kind of simple DSS, whose

system frame is formed of a dictionary data structure, an interpreter, and a compiler. The words in the dictionary reflect the functions of the Forth system, and they follow a strict call order. Tables of words are linked by the fixed dictionary structure, and the interpreter and compiler provide management and control mechanics for the dictionary.

DSS Advantages

• *shorter lead time, enhanced reliability*

DSS designers should first specify the system frame. The frame is like a system-construction tool and can improve the functional design. Modules can use hierarchical methods for reliability and ease of maintenance.

• *efficiency*

A DSS manages the system with a view to its resources, and removes the inter-layer communication barriers imposed by the hierarchical method.

• *flexibility*

In a hierarchical system, it is difficult to modify or extend the lower layers. The DSS, on the other hand, includes all the hierarchy in its tables and there are no physical layers, so adding or reducing system functions is simply a matter of expanding or deleting modules from the dictionary.

• *system programming language*

DSS permits the DMS to be designed as an interpreter and compiler (e.g., the Unix shell), which in turn can offer a programming environment built upon the system instructions (e.g., Forth).

• *open-ended program base*

This aspect of DSS can be very important to software reuse and portability.

The design of Forth systems reflects the advantages of DSS. Additionally, Forth uses reverse Polish notation (RPN) arithmetic, which is more easily handled by the computer; linear decoding, which can save memory; and irregular design.

Forth as System Language

The facts show that every language has its own application area, and Forth is the same. In the past, Forth has been used in process control and image processing, to name a couple, but I think Forth is well-suited to be a system language.

First, it describes a system structure which is simple, effective, and reliable. It provides an interpreter and compiler which are not only dictionary management tools but also user-instruction processors. The dictionary is open and composed of piled-up modules; all of the words in it form the system language, i.e., the Forth language.

Second, Forth balances the user interface with the machine interface. RPN and the irregular design method can make one feel it is difficult to master, but these very features make it easy to describe and implement a system.

To basic rule in Forth development is to retain its simple style. Some people enjoy its unique characteristics, while others reject it or try to make it into something like other high-level languages. My point is that if one is bound by his nature to be a painter, let him be a painter and not a singer; otherwise, a genius is going to be a mediocrity. Take Forth in your own direction, but follow its spirit for the best results.

Forth can support all kinds of functions, but its user interface is not good. Why won't we take advantage of other high-level language interfaces to compensate for Forth's defects, perhaps ending with Forth as a mid-level language?

Forth Processors

The introduction of Forth chips and their related Forth nucleus software convinces us of the potential advantage of such a system. In the past we saw machines that directly supported high-level languages, but design complexity, low efficiency, and limited applications brought failure. According to past experience, it seems the same problems could be encountered if we try to make a machine that supports Forth.

The chip we designed to support Forth is so simple, the gates so few, and the speed so fast that others chips cannot compare. Why? Forth itself—its style is quite different from that of other languages. It's not so much that Forth is a new kind of language, as that it is a kind of design thought and rule. As has been demonstrated, it is easy to support in hardware Forth's fundamental functions, stack-based operations, and RISC techniques. Similarly, software design is so simple, the system so flexible, and maintenance and performance so good, that other system structures cannot compare.

Problems

Integrated hardware and software design based on Forth has just begun. There are still a lot of problems to be solved. For example, Forth's current dictionary structure is limited and further study is needed to find the best solution; Forth's greatest weakness is its lack of protection, especially of stacks; the frame is the key to a DSS, so a Forth chip should support not only basic functions and stack operations, but the DSS frame, operating system, even some kind of conversion from other languages. I think both the hardware and software design of the new system should fully integrate current well-developed theory and technique.

In general, the development of Forth has just begun. When necessary, it needs to be expanded and improved, but it would be unwise to belittle Forth. It is not easy to develop an adequate enough understanding of Forth to study, use, and further develop it, but if you can grasp Forth *thought* you will gain a new understanding of the language itself.

Wu Qian has an M.S. in software engineering and fourth-generation languages from the Software Institute of the Chinese Academy of Sciences, and his thesis noted Forth's differences from traditional OS structures. He is interested in software engineering, the structure of system software, artificial intelligence, and the human-computer interface. He is designing system software for a new kind of machine that supports Forth, and is trying to create an integrated Forth system on a Forth machine.

INTERACTIVE CONTROL STRUCTURES

JOHN R. HAYES - LAUREL, MARYLAND

Forth novices sometimes type:

```
8  0  do  i  .  loop
```

into their Forth systems and then wonder why

```
0  1  2  3  4  5  6  7  ok
```

doesn't appear. The easy answer is that Forth only allows control structures to be used inside colon definitions. A more accurate answer is that Forth system implementors feel that providing interactive control is too difficult. In this article, I will describe a relatively simple way for a Forth system to provide control structures that behave consistently, whether interpreted or compiled.

Interactive control structures have many uses. For example, they can initialize an array or table:

```
create squares
100  0  do
  i  i  *  ,
loop
```

Another use for interactive control structures is conditional compilation in a file-based system:

```
Forth-79?
if  " Forth83-emulator"
load-file  then
```

`Forth-79?` is a word that tests to see if a Forth-79 Standard system is present. If so, `load-file` loads a file named Forth83-emulator that contains a Forth-83 emulator written in Forth-79.

At the 1987 FORML Conference, Mitch Bradley gave a paper appropriately titled "Interpreting Control Structures—The Right Way" [1]. His solution to making control structures interactive didn't allow compiling-words such as `allot` and `,` to be used within the control structures. For example, you couldn't initialize the `squares` table using the code above. He left this as "an exercise for the reader." I have taken up the challenge, and I provide a solution here. In the remainder of this article, I'll describe the problems found and faced in developing my solution. Source code is provided.

Problems and Solutions

The first problem in implementing interactive control is that the control structures must be compiled before they can be executed. This is easy to handle: just switch to compile mode when the beginning of a control structure (`do`, `begin`, `if`, etc.) is encountered during interpretation. But how do you know when to stop compiling and execute the compiled code? If the compiler is turned off as soon as an ending control structure (`loop`, `until`, `then`, etc.) is found, nested control structures will not work. We must keep track of the nesting level somehow.

As you grow accustomed, they may become indispensable.

I use Forth's `state` variable to count the nesting level. A `state` value of zero indicates that the system is interpreting. Control structure words such as `do`, `begin`, or `if` increment `state` and words such as `loop`, `repeat`, and `then` decrement `state`. I have added two words to my system, named `]]` and `[[` (analogous to `]` and `[`) that perform these functions. For example:

```
: do
  ]]   <do_code>
  ; immediate

: loop
  <loop_code> [[
  ; immediate
```

(`Begin`, `repeat`, `until`, `if`, and `then` are modified in a similar way.) If `do` is found while interpreting (state = 0), `]]` increments `state` and the system starts compiling (state = 1). If an `if ... then` structure appeared within the loop, `if` would increment the `state` to two and `then` would decrement it to one. When `loop` is found later, `[[` decrements the `state` back to zero and interpretation is resumed. `[[` detects this transition and executes the compiled code.

The code within the control structure is transient and must be compiled into some temporary location. The obvious location is the end of the dictionary at `here`. The problem with this is that words such as `allot` and `,` couldn't be used with the control structures. For example, in the `squares` example, 'comma-ing' in the squared values would overwrite the `do ... loop` code and almost certainly crash the system. A separate compile buffer would solve this problem.

In an earlier article on local variables (*FD* XI/1), I described a way to add compile buffers to a Forth system. I changed the dictionary pointer variable `dp` (called `h` in some systems) into a colon definition to add a level of indirection. Since `allot`, `here`, and ultimately the entire compiler are defined in terms of `dp`, changing the value returned by `dp` can redirect the compiler to another region of memory.

```
variable regionptr
: dp
```

EuroFORML'90

Large Systems
(Forth in Control in the 1990's)

October 12-14th 1990

Call For Papers

Suggested Subject Headings Are:

Connectivity, Multi-processor Systems, Distributed Systems, Project Management, Team Programming, Techniques and Tools.

Please let us know as soon as possible if you would like to speak. Abstracts should be submitted by August 12th and papers, camera ready, by September 12th.

The venue:

The Potters Heron
Ampfield
Hampshire

Situated on the edge of the picturesque New Forest, this extensive thatched hotel offers all modern facilities. The famous Broadlands and Beaulieu stately homes are only a short distance away as are the award winning Exbury and Hillier Gardens. Sample the ancient splendour of the historic Winchester and Salisbury Cathedrals or try a traditional New Forest Cream Tea.

A Demonstration and Exhibition area is available- please contact the Conference Organiser for and information sheet.

All communications to:

The Conference Organiser
EuroFORML'90
133 Hill Lane
SOUTHAMPTON SO1 5AF

Tel: (+44) (703) 631441

```
  regionptr @ ;
\ ( -- addr ) Return next free
\ location in allocation
\ region

: allocatefrom
  regionptr  !  ;
\ ( addr -- ) Select an
\ allocation region
```

New compile buffers are created by the following defining word:

```
: region
  create here
  cell+  ,  allot  ;
\ ( size -- )  Create
\ allocation region
```

The control structure compile buffer is created as:

```
200 region compileregion
```

Stdregion is the built-in standard dictionary allocation region.]] and [[automatically switch the compiler between compileregion and stdregion. The diagram in Figure One summarizes the states the Forth system can occupy. In the topmost state in the figure, the system is interpreting and dp refers to the end of the dictionary. Starting a colon definition sets the state to one, and control structures behave conventionally in the leftmost group of states. If]] is called via do, if, etc. while interpreting, the system enters the rightmost group of states and starts compiling into the compile buffer. Further control structures will nest properly. When the final loop, then, etc. calls [[, it switches back to the standard allocation region, executes the code in the compile buffer, and resumes interpretation.

The Code

The source code is shown in the accompanying listing.]] first checks to see if the system is interpreting (i.e., is this the 0-1 state transition?). If it is,]] switches dp over to the compile buffer. Here, which now indicates the next free location in the buffer, is saved in a variable named compilebuffer. Since the code in the compile buffer will later be passed to execute, we need to create a code field in the buffer. The line labeled NON-PORTABLE does this in a system-dependent way. For example, in an indirect-threaded-code system, start: might ' (tic) a known colon definition and copy the code field:

```
['] <known_colon_def> @ ,
```

In a subroutine-threaded system, start: would be a no-op. Finally,]] increments state.

[[decrements state. If this brings the system to the interpret state, [[knows there must be something in the compile buffer to execute. An exit is added to the buffer, dp is switched back to the standard allocation region, the code pointed to by compilebuffer is executed, and the

(Continued on page 41.)

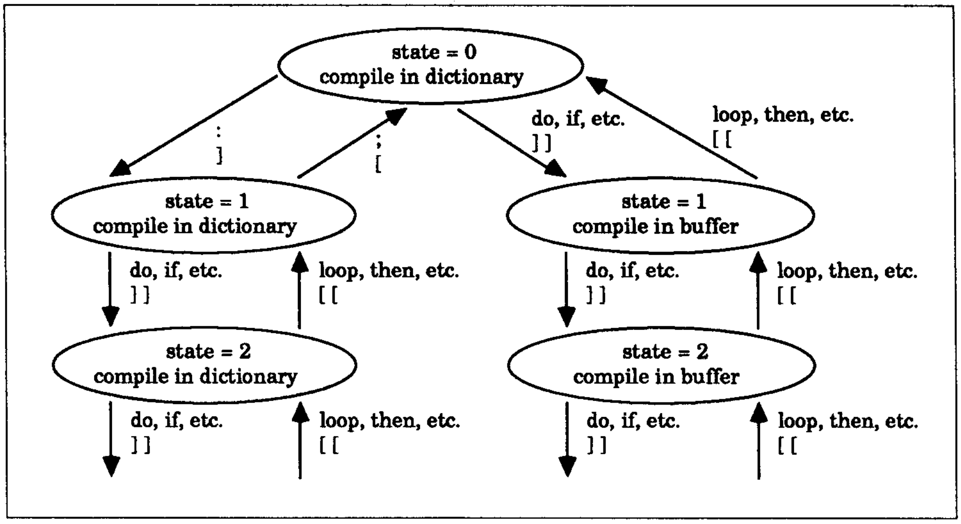

Figure One. System state diagram.

METACOMPILE
BY DEFINING TWICE

CHESTER H. PAGE - SILVER SPRING, MARYLAND

The basic idea behind this metacompiler is to use a *host* Forth in out-of-the-way memory to define a new, *target* Forth dialect in the normal memory area, using normal Forth definitions. The addresses of the components of new colon definitions are taken from the target (when available), otherwise from the host (if possible). For components not defined in `HOST` and not yet defined in `TARGET`, a 0000 is compiled and announced. After one pass through all the target definitions, the target words all occupy the correct amount of space and are located at their final addresses. The temporary host component addresses and 0000s are to be replaced by target addresses on a second pass through the set of definitions. Actually, each definition is completely overwritten, but with unchanged addresses where the first pass gave the desired final addresses.

The major requirement for redefining with overwrite is maintaining the integrity of the links between target words, so that the target vocabulary can be searched properly when compiling definitions. The first problem is having input-stream words overwrite link fields; this is avoided by not using the upcoming dictionary area for `WORD` input. A special area (`STREAM`) is used, and its content is transferred to the dictionary area when appropriate.

The second problem involves new linkages. The second defining pass starts with the dictionary pointer reset to the origin of the target; if the target vocabulary is still called `TARGET`, the first word will link to the previous last word, making a circle that prevents proper searching. This is avoided by setting up a dummy vocabulary for target words in the second pass. Linkages in `DUMMY` will run from the latest word redefined, down to the first word. Linkages in `TARGET` will be from the final word down through all words defined only once, then down through the words that are also in `DUMMY`, because the dummy links are repeats of the original `TARGET` links!

The first pass has some improper component addresses, but each word is at its correct location, so the second pass will find a correct address for each component—including any that were not in existence when they were needed in the first pass. Thus, host addresses and 0000s will be replaced by the desired target addresses.

One problem...is to keep CONTEXT and CURRENT under control.

All colon words are subject to a second pass; assembled primitives need not be redefined, since all components are correctly located on the first pass. Where a primitive makes a reference to an address in another primitive, or to the storage cell of a Forth variable, labels are used and label addresses are not substituted until the end of the first pass, so they do not need correcting. Skipping the screens of primitive words during the second pass requires adjusting links and the dictionary pointer across the skip. For example, consider a group of screens of primitives to be skipped; when it is reached in the second pass, identify the last word in these screens as `LATEST` (so that the next word defined will link to it—in a multi-threaded system, this must be done for each thread) and set the dictionary pointer to the first byte of the header of the first word on the following screen, so that the overwriting will start in the proper location.

It is convenient to group primitives into successive screens to simplify this maneuver. For example, in my definition of this metacompiler, the first ten screens are the basic primitives of Forth, followed by four screens of constants and variables, 27 screens of colon words, and finally nine screens of primitive words. Only the 31 screens in the middle are subjected to a second pass.

During each pass, only host words are to be executed, but target words are to be compiled. Making `TARGET` the `CURRENT` vocabulary and `HOST` the `CONTEXT`, solves most of the problem. All defining words, such as the colon, are selected from `HOST`. Immediate target words, however, would be executed. This is avoided by making `IMMEDIATE` include `SMUDGE`, so that all immediate words in `TARGET` are automatically passed over because of a smudge. A problem then arises when an immediate word is to be compiled by `[COMPILE]`! This "Catch-22" situation is resolved by substituting a special version of `[COMPILE]` in `HOST`: the `SEEK` component of `[COMPILE]` is modified to change the smudge status of each input-stream word, so that `[COMPILE]` can now find only smudged words. There are times when `'` is needed for finding a target word—`TARGET` must be the `CONTEXT`, but the host `'` is to be executed. This is handled by smudging the target definition of `'`.

If a defining word that is missing in `HOST` is to be used in `TARGET`, a suitable version of the defining word must be added to `HOST`. This can be done as an *ad hoc* addition from the metacompiler. For example, my minimal host does not have `ARRAY` as a defining word (intentionally), but my target needs an array `VOC.LIST`.

So a version of ARRAY is temporarily added to HOST. A modified version of VOCABULARY, called $VOCAB to avoid confusion, is also used for creating new vocabularies in the target *during compilation.*

The ultimate machine to be defined is called FORTH, and this is defined as a vocabulary in TARGET. Also defined in TARGET is the vocabulary NEW with one word in it, to test the compiler.

Compiling 0000 for any word not found is achieved by substituting a special *ad hoc* version of INTERPRET in HOST. (These modifications to HOST are not permanent—they are not made in HOST, but are substitutions put into HOST in memory by the metacompiler.)

Since the final colon words must have TARGET generic execution procedure (rather than HOST code field references), the host's : (colon) must be redefined before the second pass by substituting the target DOCOL for the host DOCOL in the host colon. Similarly, the target's EXIT must be substituted into the host's ; (semicolon). All words compiled by immediate words in HOST must be replaced by their TARGET counterparts so that target addresses will be compiled. Examples of compilees to be substituted are ?BRANCH, BRANCH, (DO), (LOOP), (."), etc.

All the other special generic execution procedures (DOVAR, DODOE, DOCON, etc.) must also be replaced in the second pass, and all of the addresses compiled by immediate compiling words must be updated before the second pass.

Since both defining passes are operations in HOST, vocabulary reference changes are made in the host's CONTEXT and CURRENT. After the second pass, the SET.CONTEXT (by which calling a vocabulary sets it to CONTEXT) must be changed to refer to the target's CONTEXT. The vocabulary specified to be searched next after a word is not found must also be changed *in each secondary vocabulary* to FORTH. BASE, DP, and FENCE must be properly initialized, the topword pointer(s) set in FORTH, and the starting word(s) at the origin linked back as the first word(s) in FORTH.

The final step is to unsmudge all smudged words and call for a cold start of Forth. This newly defined Forth can then be saved to disk.

```
screens SCR # 1
\                                                    13MAR89CHP
HEX
E6 CONSTANT N
N 2- CONSTANT DIV
N 8 + CONSTANT IP
IP 3 + CONSTANT W
W 2+ CONSTANT XSAVE
E0 CONSTANT TEMP
E2 CONSTANT TEMP1
FDED CONSTANT COUT
FC22 CONSTANT VTAB
FC58 CONSTANT HOME
FD35 CONSTANT INCH
900 CONSTANT ORIG
57B CONSTANT CH
25 CONSTANT CV      -->

screens SCR # 2
\                                                    13MAR89CHP
28 CONSTANT BASL
FBC1 CONSTANT BASCALC
C054 CONSTANT PAGE1
C055 CONSTANT PAGE2
BE03 CONSTANT BI.PARSE
BE0C CONSTANT ERR.PRINT
BE6C CONSTANT PN1ADD
BF00 CONSTANT MLI
FD8E CONSTANT CROUT
VARIABLE VOC.SAVE
VARIABLE LFLAG
VARIABLE DP.HOLD   -->
\ These ROM and ProDOS system constants provide for using
\ built-in operations, such as disk input and output

screens SCR # 3
\                                                    13MAR89CHP
: &NUMBER ( here--d true;here false) 0 0 ROT DUP 1+ C 2D =
   DUP >R + -1 BEGIN DPL ! CONVERT DUP C@ DUP 2E =
              WHILE DROP 0 REPEAT
   BL = IF DROP R> IF DNEGATE THEN 1
        ELSE R> DROP >R DROP DROP R> 1- 0 THEN ;
: &INTERPRET BEGIN SEEK ?DUP
    IF STATE @ + 0= IF , ELSE EXECUTE THEN
    ELSE &NUMBER
       IF DPL @ 1+
         IF [COMPILE] DLITERAL ELSE DROP [COMPILE] LITERAL THEN
       ELSE ." New word used as component; 0000 compiled" CR
          DROP 0 ,
       THEN
     THEN
              AGAIN ;    -->

screens SCR # 4
\ Substitute &INTERPRET for INTERPRET                13MAR89CHP
' &INTERPRET DUP ' QUIT 12 + ! ' LOAD 40 + !
\ Create dummy ARRAY and VOC.LIST in host source
: DOARR SWAP 2* + ;
: ARRAY CREATE DUP -2 ALLOT [ ' DODOE 2+ ] LITERAL ,
   [ ' DOARR 2+ ] LITERAL , , 0 DO 0 , LOOP ;
7 ARRAY VOC.LIST
\ Provide vocabulary-creating word for use while compiling
: &VOCAB CREATE -2 ALLOT [ ' DODOE 2 + ] LITERAL ,
  [ ' SET.CONTEXT 2+ ] LITERAL , A081 , HERE 2- DUP DUP DUP
  [ ' FORTH 6 + ] LITERAL , LAST
  1 BEGIN DUP VOC.LIST @ WHILE 1+ DUP 7 =
   ABORT" Too many vocabularies" REPEAT VOC.LIST ! ;
5 LOAD
\ NOTA BENE: Need LOAD instead of --> to activate use
\ of &INTERPRET in screen interpretations
```

```
screens SCR # 5
\                                                     13MAR89CHP
 39 LFLAG !    \ Screen $38 is last colon-word defining screen
\  It ends in LFLAG @ LOAD -- on first pass loading proceeds
\ to the second batch of primitives; on the second pass
\ loading will be diverted to the wrao-up screens $A/B
60 ' IMMEDIATE 6 + C!  \ Makes IMMEDIATE also SMUDGE
\ Rename host FORTH as FARTH
' FORTH >NAME 2+ 41 SWAP C!
VOCABULARY FIRTH
FIRTH DEFINITIONS
FARTH
ORIG DP !
ASSEMBLER
CLEAR.TABLES
0D LOAD  \ First screen fo primitive definitions


screens SCR # 6
\ Prepare for second pass, redefining with overwrite 13MAR89CHP
\ Redefine ARRAY to avoid erasing arrays defined in first
\ pass, e.g., VOC.LIST holding names of vocabularies
\ defined in first pass
FARTH DEFINITIONS
: ARRAY CREATE DUP -2 ALLOT [ 0 ] LITERAL , [ 0 ]
   LITERAL , , 2* ALLOT [ 0 ] LITERAL , ;
\ Now substitute FIRTH addresses in this definition
FIRTH ' DODOE 2+ FARTH ' ARRAY 0E + !
FIRTH ' DOARRAY 2+ FARTH ' ARRAY 14 + !
FIRTH ' EXIT FARTH ' ARRAY 20 + !
\ Provide for immediate compiling words to compile FIRTH
\ addresses instead of the locked-in FARTH addresses
FIRTH ' ?BRANCH DUP FARTH ' IF 4 + ! ' UNTIL 4 + !
FIRTH ' BRANCH DUP FARTH ' ELSE 4 + ! ' AGAIN 4 + !
-->


screens SCR # 7
\                                                     13MAR89CHP
FIRTH ' (LOOP) FARTH ' LOOP 4 + !
FIRTH ' (+LOOP) FARTH ' +LOOP 4 + !
FIRTH ' (DO) FARTH ' DO 4 + !
FIRTH ' LIT FARTH ' LITERAL C + !
FIRTH ' (DOES) FARTH ' DOES> 4 + !
FIRTH ' (.") DUP FARTH ' ." 0A + ! ABORT" 8 + !
FIRTH ' QUIT FARTH ' ABORT" 14 + !
FIRTH ' SP! FARTH ' ABORT" 10 + !
FIRTH ' VOC.LIST DUP FARTH ' &VOCAB 3C + ! ' &VOCAB 65 + !
FIRTH ' DODOE 2+ FARTH ' &VOCAB C + !
FIRTH ' DOCON FARTH ' CONSTANT 8 + !
FIRTH ' DOVAR FARTH . CREATE A5 + !
-->


screens SCR # 8
\                                                     13MAR89CHP
\ Provide for [COMPILE] to compile SMUDGED IMMEDIATE words
\ Define *SEEK to find unsmudged words by smudging the sample
: *SEEK BL WORD COUNT UPPER THREAD! STREAM DUP 20 TOGGLE
  [ ' FIRTH 6 + ] LITERAL THREAD @ 2* + @ (FIND) ;
\ Substitute this into [COMPILE]
' *SEEK ' [COMPILE] 4 + !
\ Next step is at the HEART OF THE METACOMPILER
\ Redefine FARTH's : to force FIRTH (in place of DUMMY) as
\ context during compilation of the definition of a colon-word
: CLASS R> @ 2+ LAST NAME> ! ;
: : ?EXEC CONTEXT @ !CSP CREATE FIRTH ] CLASS DOCOL ;
FIRTH ' DOCOL FARTH ' : 12 + !
FIRTH ' EXIT FARTH ' ; 0A + !
-->
```

About Vocabularies

One problem in developing a program like this metacompiler, which is operating in several vocabularies, is to keep CONTEXT and CURRENT under control. In the case of defining vocabularies during compilation, the following routine was used:

```
CONTEXT @  CURRENT @
&VOCAB FORTH
&VOCAB NEW
FIRTH
NEW DEFINITIONS
: NEWTEST
  ." NEWVOC" ;
CURRENT !
CONTEXT !
```

The other place requiring a juggle is in defining the colon words. Forth normally makes the CURRENT vocabulary the CONTEXT during a definition, so that components will be found in the appropriate vocabulary. In this case, we want FIRTH to be the context during the redefinition phase in which DUMMY is CURRENT. Screen eight redefines the host colon to accomplish this.

My realization of this concept comprises 12 screens of metacompiler instructions, followed by the definitions of the desired Forth. In my case, these definitions take 53 screens. I wish to emphasize that these definitions are all in standard forms, with no special considerations for the metacompiler (except for the use of &VOCAB when defining additional vocabularies, and this could be avoided by using VOCABULARY as its name); actually, these definitions were the Forth language description of my assembly program for generating my Forth. The host is based on a minimal version of Forth that has to meet only two requirements: it must be able to load screens, and it must support an assembler. The minimal version with the assembler compiled onto it is the HOST of the above discussion. In my Apple][e, HOST lies from $5000 up, leaving the 18K from $800–$4FFF available for the Forth being compiled.

Machine-Specific Considerations

The routines mentioned above are illustrated by the following actual realization of a metacompiler, written for use with the Apple][family of computers. These screens are only for illustrating the prob-

(Continued on page 40.)

```
screens SCR # 9
\                                                                13MAR89CHP
VOCABULARY DUMMY
DUMMY DEFINITIONS
\ Set linkages to last primitives on screen 22
\ and dictionary pointer to first words on screen 23
FIRTH ' @ >LINK FARTH 0 THREAD1 ! LATEST !
FIRTH ' EXIT >LINK FARTH 1 THREAD1 ! LATEST !
FIRTH ' NEGATE >LINK FARTH 2 THREAD1 ! LATEST !
FIRTH ' SETUP >LNK FARTH 3 THREAD1 ! LATEST !

FIRTH ' ORIGIN >LINK FARTH DP !    \ Leaves FARTH as CONTEXT
\ After all colon-words redefined, return to screen $A
\ for wrap-up
A LFLAG !      \ After second pass, screen $38 goes to screen $A
ASSEMBLER
16 LOAD     \ Go to screen 22 to start second pass


screens SCR # 10
\                                                                13MAR89CHP
FARTH DEFINITIONS
HEX 8800 DP !   \ To avoid overwrites
: (UNSMUDGE) @ BEGIN DUP 2+ DUP C@ 20 AND IF CR DUP ID.
  DUP 20 TOGGLE THEN DROP @ DUP @ A081 = UNTIL DROP ;

: UNSMUDGE 4 0 DO ['] FIRTH 6 + I 2* + (UNSMUDGE) LOOP ;
: CHANGE.SET [ FIRTH ' SET.CONTEXT FARTH 2+ ] LITERAL
   1 BEGIN OVER OVER VOC.LIST @ ?DUP
   WHILE NAME> >BODY ! 1+ REPEAT DROP DROP DROP ;
: NEW.SEARCH ['] FORTH 6 + 2 BEGIN OVER OVER VOC.LIST @ ?DUP
  WHILE NAME> 0E + ! 1+ REPEAT DROP DROP DROP ;

CHNAGE.SET
NEW.SEARCH
-->


screens SCR # 11
\                                                                13MAR89CHP
\ Special problem - replace the ' in ['] with the FORTH
\ version, remembering that both ' and ['] are smudged !
  *SEEK ' DROP *SEEK ['] DROP 4 + !
\ Make FIRTH both CONTEXT and CURRENT in FARTH
' FIRTH 6 + DUP CONTEXT ! CURRENT !
\ Initialize BASE, DP, and FENCE
A BASE ! DP.HOLD @ DP ! \ DP.HOLD stored at end of screen 64
\ Set topword pointers, and set 0000 as next search after FORTH
  ' FIRTH 6 + ' FORTH 6 + 8 CMOVE    0 ' FORTH 0E + !
\ Make FORTH the CONTEXT and CURRENT in FIRTH
' FORTH 6 + DUP CONTEXT ! CURRENT !
' FORTH 4 + DUP DUP DUP ' NEXT >LINK ! ' CLIT >LINK !
   ' LIT >LINK ! ' I >LINK !
CONTEXT @ PAD 8 CMOVE PAD 4 LARGEST FENCE ! DROP
UNSMUDGE     COLD


screens SCR # 12
DESCRIPTION OF SCREENS 12 THROUGH 64                             13MAR89CHP

Screens 12/21 hold assembly definitions of primitives
22/25 hold various constants and variables; these must be
defined under the ASSEMBLER vocabulary because several
labels are assigned; and redefined to convert the addresses
of the generic execution procedures for constants and variables
to addresses in FIRTH in place of the original addresses in
the HOST
25 ends with FARTH to leave the ASSEMBLER vocabulary
26/55 hold colon definitions; 55 ends in LFLAG @ LOAD
56 starts with 6 LFLAG ! to take effect at the end of 64
56/64 hold the second batch of primitive definitions
64 ends in
END FARTH HERE DP.HOLD !
LFLAG @ LOAD
```

Many FIG Board Terms Drawing to a Close
CALL FOR NOMINATIONS

The nominating process for the selection of officers for the 1991 FIG Board of Directors is starting. The candidates elected to the five available director positions will be able to serve a three-year term, with the possibility of reelection thereafter.

To be considered for nomination or to obtain a nomination by petition, you should carefully read these instructions. The nomination and subsequent election processes take place as proscribed by our Bylaws. As the following extract from Article VIII, Section 1 of the Bylaws indicates, open elections are made possible by the timely completion of steps stretching over at least a five-month time period. The first step has already been taken by the current Board of Directors through the appointment of Mike Elola and Jack Brown to the Nominating Committee for this election year.

FROM THE BYLAWS...

(a) Nominating Committee. The Board of Directors shall appoint a Nominating Committee composed of at least two Directors to select qualified candidates for election to vacancies on the Board of Directors at least 120 days before the election is to take place. The Nominating Committee shall make its report at least 90 days before the date of the election, and the Secretary shall provide to each voting member a list of candidates nominated at least 60 days before the close of elections.

(b) Nominations by members. Any 25 Members may nominate candidates for directorships at any time before the 90th day preceding such an election. On timely receipt of a petition signed by the required number of Members, the Secretary shall cause the names of the candidates named on it to be placed on the ballot along with those candidates named by the Nominating Committee.

(c) The Corporation shall make available to all nominees, an equal amount of space in *Forth Dimensions* to be used by the nominee for a purpose reasonably related to the election.

(d) Should a petition be received, a ballot process will be provided to the voting membership. Otherwise, the Secretary shall cast a unanimous ballot for the candidates as proposed by the Nominating Committee.

OBTAINING A NOMINATION

The Nominating Committee selects candidates for the ballot. FIG members who wish to become candidates this way should submit a letter requesting consideration by the Nominating Committee (c/o FIG office) before the deadline.

Alternately, a potential candidate shall obtain at least 25 signatures from FIG members and send this petition to the FIG Secretary (c/o FIG office) before the deadline. The names of qualifying candidates are placed directly on the voting ballot.

The deadline for submitting either nominating petitions or letters requesting consideration by the Nominating Committee is August 31, 1990. Send these items to the FIG office at P.O. Box 8231, San Jose, CA 95155.

(If the Secretary does not receive any nominating petitions by August 31, 1990, then the Secretary will cast a unanimous vote for the candidates selected by the Nominating Committee. In such a case the membership at large will not receive voting ballots.)

The next important date after August 31 of this election year is September 14. It is the deadline for candidates to submit their candidate statements so that they can appear in the November-December issue of *Forth Dimensions*.

Ballots will be included in the November-December issue of *Forth Dimensions*, if necessary. The voting ballots must be returned to the FIG office by December 31, 1990. The newly elected directors assume their duties the following day.

The following FIG members nominate <candidate-name> to the FIG Board of Directors.

MEMBER NAME (Please Print)	MEMBER SIGNATURE	MEMBER NUMBER
<name1>	<name1>	<number1>
<name2>	<name2>	<number2>
<name3>	<name3>	<number3>
.	.	.
.	.	.
.	.	.
<name25>	<name25>	<number25>
.	.	.
.	.	.
.	.	.

Nominating Petitions must be worded as is shown

BEST OF GENIE

GARY SMITH - LITTLE ROCK, ARKANSAS

News from the GEnie Forth RoundTable—One of the most common questions asked on the GEnie Forth RoundTable Bulletin Board, if not *the* very most common, is, "What Forth should I get for my (fill in computer/chip/application)?" This is not an unreasonable query—even from a seasoned Forth programmer—when one considers there were 145 files posted in our on-line Library #4, "Public Domain and Sample Systems," the evening I was writing this column. Expect one or two additions by the time this piece is delivered to your mailbox. While that means there are many offerings to filter through for the one that might best serve your needs, it also means there is probably a kernel available for your needs. Keep in mind, this does not include the splendidly supported products provided by systems vendors and embedded-board and programmed-chip vendors.

First let's examine some specific requests and the responses they elicited. Then I will close this with a sampler of some of the public-domain kernels available in the library, including some rather unique ones.

Topic 7:
Which Public-Domain Kernel?

From: Alex Kozak
Re: 8080 fig-FORTH

I'm looking for an 8080 fig-FORTH with source in CP/M ASM.

To: Alex Kozak
From: Gary-S

Alex, there are two kernels on GEnie (which I know are also available on xCFBs) you may wish to consider:

- GEnie #1418. FORTHLIB.ARC is a 79-Standard kernel modeled after Glen Haydon's *All About Forth.*
- GEnie #701. UNI4TH80.ARC is Uniforth's public-domain sampler, but is quite complete.

If neither of these meets your needs, I can upload to GEnie (and it will be ported to the xCFBs if you can't get on GEnie) a public-domain fig-FORTH written for a Kaypro II you should be able to run as-is on your Ozzie. I had to massage it some to get it to go on a Bondwell-12, but I don't think you will have the same problem with your O-1. —Gary

It's hard to imagine a computer for which there is no Forth.

From: Ben Combee
Re: Forth for the IBM

Do you know about a good public-domain Forth for an IBM XT-compatible that comes with (or has available) words to access EGA graphics? I have used F83 (dated rather a ways back) but did not like its interface.

Also, does anyone know of a good interactive tutorial for Forth that will run on the same system? I have seen similar programs for Turbo Pascal and the other "in" languages. While I have read both editions of *Starting Forth*, I still am having problems getting anything done.

To: Ben Combee
From: Sysop (ECFB/Shifrin)

You may be interested in checking out Tom Zimmer's F-PC. It's built on F83, is text-file based, and has numerous extensions including a number for EGA and VGA graphics. Look for FPC225-1.ZIP through FPC225-5.ZIP. If you can't find it locally, you're welcome to log in here to download it. You can also order the base system from the author for $25.

Re: "Also, does anyone know of a good interactive tutorial for Forth that will run on the same system?"

The two Forth boards and GEnie have an on-line tutorial based on this implementation. It's too big to network, but you're welcome to stop by and download the lessons or participate in that conference. Good luck!

From: Bob Bileski
Re: What Forth

I would like to start learning Forth by picking a system that is current and that I can grow with. After downloading a number of Forths, I felt F88 by Zimmer was as close to a friendly language as I've seen. I've read that FPC225 is better because it is based on Zimmer's Forth. I guess at this point I'm totally baffled by the millions of Forths available, as well as the number of extensions and fixes. Should I start with FPC225, F88, TIL, Harvard, LMI, F83, F83X, Zimmer's, etc... Should I use one from the BBS or can I purchase a Forth and all the attachments from one source? I spoke to Offete about the FPC225 (total package) and all I really came away with is "...very, very good. Send check. $75. You'll like it very much."

I give up. Perhaps you can advise what would be the best long-term Forth to purchase. Does anyone sell Forth diskettes and tutorials, so I can minimize my BBS phone charges? I suppose the vast amount of unstructured Forth products makes many shy away from this language. Maybe I'm missing something. I'd appreciate any info you might give me... Thanks.

To: Bob Bileski
From: Steve Palincsar
Bob, what Forth to get should be, in part, determined by what you want it for, what you intend to do with it, and the state of your budget. For example, LMI and Harvard Softworks—to name only two—both have excellent, very complete products suitable for professional-level work, at corresponding prices. With either of these systems, you'll get a fine product (actually, LMI has a number of fine products, both 16 and 32 bit, for both DOS & OS/2). There are any number of public-domain Forths out there, as you've discovered, some of which have a lot of features but all of which seem to have a chaos of versions, incompatible extensions, etc. There are systems that fit into about 4K, and there are systems that seem to take about 400K. I think some of Jerry's text files here, including "which4th" might be of help to you, but the most important thing is to decide what you want to do with it. If it's presently learning something about the language, to get a feel for it, you might want to start with a fairly simple (and limited) public-domain system.

Unless, of course, 400+ dollars is a mere bagatelle...

To: Steve Palincsar
From: Bob Bileski
I see what you mean. I originally wanted to use Forth for an embedded controller project. After downloading a number of Forths, I realized that ROMming a Forth kernel is not as easy as I was led to believe. I think I will use F-PC Forth for my learning process at this time. As for the controller, I'll use the Intel 80c52 Basic microcontroller; it has all the EPROM burning, machine calls, etc. that I need built-in. As far as being slower than Forth, that won't be a problem. I looked at ZenForth, but couldn't find documentation to get me going. Perhaps I really need to learn what I'm doing before I can make any intelligent decisions in regards to using Forth for control projects (non-commercial).

From: DanMiller
Also check Pygmy for an 8088 version of cmForth, which is a good minimal system for embedded control. Pygmy, translated by F. Sergeant is available on GEnie, and includes a small metacompiler to generate code. cmForth for the RTX processor is ROMmable. Pygmy includes notes on regeneration.

From: Gary-S
Re: "Does anyone have a public-domain x386, x286, or 680xx Forth system they can *easily* send me over the network to run on either kind of Sun workstation?"

Sun workstation => unix => Mitch Bradley's cForth.

Easy solution: send $50 to P.O. Box 4444, Mountain View, California 94040.

From: Eric Therkelsen
I like what I've seen of Forth (F83 and the Inner Access S8), and intend to make it my principal language for in-house projects (and out-house, if I can sell it). Can anyone recommend a Forth package for the PC (MS-DOS) that:

- has a fairly complete set of extensions for string handling, file access, and math, including floating point;
- is more or less in the mainstream—that is, whose extensions aim more or less in the direction Forth seems to be headed;
- has been around long enough to be stable and has a good user base.

Wish list:

- allows precompiled function libraries;
- interfaces to the hardware either via DOS functions (rather than using ROM-BIOS services) or via redefinable words, so it will run on the Z100 as well as on AT-compatibles.
- internals available, metacompiler, etc. I don't want to *have* to use these, but I like to be able to if necessary. Also, they're fun to play with late at night.

Any help would be greatly appreciated.

To: Eric Therkelsen
From: Steve Palincsar
HS/Forth will do all that you ask, including being able to use precompiled C libraries. It includes a metacompiler and all the string functions described in Kelly & Spies' *Forth: a Text and Reference*, and is as solid and complete a system as you could wish for.

To: Ben Combee
From: Steve Palincsar
Ben, if you're looking for a public-domain Forth for the PC that has more pizazz than F83, F-PC is your logical choice. By the way, the most recent version of F83 *is about a 1984 date*. You can get F-PC from the Forth Interest Group, from C.H. Ting's Offete Enterprises, or direct from Tom Zimmer, I believe, if you can't find a BBS that has it.

Ported from uucp =>
Looking for versions of Polack's FPT-F83.ARC and F83 that run together. I would like an 8087 interface in Forth. I tried F-PC and the program was too big to load on my 256K Sanyo MBC 555 (IBM-compatible) machine with two 360K drives. —Jina Chan

To: well!gars@LLL-WINKEN.LLNL.GOV
sphinx@milton.u.washington.edu
Re: 8087 interface
We are currently trying to port Forth archives to Simtel20. I think F83 is already there. I am going to post your inquiry and my reply to ForthNet for confirmation.

uunet!swbat!texbell!ark!lrark!glsrk!gars
(My own unix sys—Gary)

From: Stephen Minton
Subj: F83 (L&P) CP/M version
I need to download the latest available version of F83 (L&P) for CP/M (I have 1.00). Where would I find this, and was it updated to 2.10 like the PC version? Thanks!

From: Gary-S
To: Stephen Minton
There are several upgraded L&P F83s available on GEnie and the xCFBs for CP/M users. They did not follow the 2.10 version notation, but have such additions as full-screen editors and alphabetized word lists. You may prefer to look at the Silicon Valley FIG (John Peters) disks—posted on GEnie and the xCFBs—and roll your own. There is also the Australian M-20 you may wish to consider, which accepts text files.

From: Scott Roberts
Re: Forth source for 8088
Does anyone know of a Forth system that would be suitable for storing in ROM of a controller based on the Intel 8088? If I could get an assembler source listing, etc.,

I could modify it to suit my system. Thanks in advance.

To: Scott Roberts
From: Jerry Shifrin

You might want to look into the Zen-Forth files available on the xCFBs, GEnie, and perhaps even get in touch with Martin Tracy (their author). I understand that Zen has been successfully ROMmed into various environments.

From: Ian Green
Re: cmForth

Anybody seen a cmForth system for DOS? I currently only have F-83 in my tools directory and would like a copy of cmForth to fiddle with.

A brief look at some of the kernels posted to library four will satisfy most everyone that there is likely a public system available for your computer. It is but a modem call away. Some of the files for various computers include the following :

Number: 1710
Name: POCKET4.SIT
Address: C.Heilman
Description: This Macintosh StuffIt file contains the Pocket Forth vers. 4 application and Deck Accessory. Also includes a number of extension source code files.

Number: 1647
Name: MX20.ARC
Address: L Collins
Description: Text-based Forth for CP/M from Lance Collins MM FORTH

Number: 1596
Name: GSFORTH.BQY
Address: D.M.Holmes
Description: This contains the main file in the Apple GS Forth Demo Package.

Number: 1541
Name: STFORTH.ARC
Address: ECFB
Description: Forth-83 version 1.0 for Atari ST distributed by the San Leandro Computer Club.

Number: 1846
Name: PURPLE.FORTH.BQY
Address: J.Purple
Description: FIG for the Apple II series. This file has been compressed using an Apple-specific compression technique.

Variety is also ever present, as evidenced by this incomplete list of MS-DOS-specific files. Note, I didn't even bother with the FIG, L&P F-83, and Uniforth sampler.

Number: 900
Name: BBL.ARC
Address: Green.ECFB
Description: This is Roedy Green's unique gift to the Forth world. It is a very fast 32-bit public-domain (except for military use) Forth that uses multiple pointers.

Number: 1964
Name: F-PC35-1.ZIP
Address: D.Ruffer
Description: This is version 3.5 of Tom Zimmer's F-PC for MS-DOS computers. F-PC is a turbo-like environment for MS-DOS Forth users. F-PC comes with an amazing array of support files, and is easily the banner system for the maxi-Forth proponents.

Number: 1939
Name: PYGMY12.ARC
Address: F.Sergeant
Description: Here is Pygmy Forth version 1.2. It is faster, more accurate, and more compatible. It automatically sets up for color or monochrome monitors. Turnkey is much easier with DEFERed BOOT. As always, it includes full source code, assembler, metacompiler, documentation, and a *Starting Forth*-compatibility file. Pygmy is representative of the minimalist-Forth approach. Gary

The arguments, from those running Unix and Unix-like environments, that there were no good Forths available for them, vanished sometime back with Alan Pratt's public-domain cForth and Mitch Bradley's supported CFORTH-83. Here are two more-recent entries for that arena.

Number: 2003
Name: BOTFTH68.ARC
Address: Gary-S
Description: botForth is another effort at a clean, universal, minimal Forth kernel. This is the 2/90 port to MC68K CPUs. There are no associated Makefiles.

Number: 1944
Name: TILE
Address: Gary-S
Description: Mikael Patel's public-domain F83—written in Unix shells.

Language is not necessarily a barrier to using Forth, either. Witness these German and Russian Forth versions.

Number: 1576
Name: VOLKS4TH.ARC
Address: K Schleisiek
Description: Documentation for this is in German, so it's pretty hard for me to tell you anything about it, except that it's for a PC. Here is the Copyright: Die Programme und die zugehrigen Quelltexte knnen frei verwendet werden. Das beinhaltet die Weitergabe und Nutzung der Programme und gilt selbstverstndlich auch fr Applikationen, die auf volksFORTH aufgebaut sind. Das Handbuch unterliegt dem Copyright (c) 1985 - 1988 Klaus Schleisiek, Ulrich Hoffmann, Bernd Pennemann, Georg Rehfeld und Dietrich Weineck.

Number: 1737
Name: ASTRO4TH.ARC
Address: D.Ruffer
Description: AstroFORTH, from Russia, is a software development system for design software of different kinds. AstroFORTH includes the Forth-83 language standard, extended by a number of service procedures and software packages providing users with additional facilities. AstroFORTH may be used on IBM PC XT/AT-compatible computers, equipped with the i8086/i8088 microprocessors. The system operates under the control of MS-DOS. For the system to work, one needs 128K main memory, a floppy/hard disk drive, a color/monochrome display controller, and (if required) a printer.

I will end this session with evidence it would be difficult to imagine a computer environment for which there is no Forth. An argument has raged about the Forth-like qualities of PostScript, Adobe System's de facto *graphics system. There are many features of PostScript that smack strongly of a Forth heritage. In answer to the arguments, Mitch Bradley created a skeletal version of Forth in—yep!—PostScript.*

Number: 1995
Name: PSFORTH.02.90
Description: In response to messages from Doug Philips regarding why "PostScript does *not* qualify as Forth," Mitch Bradley

(Continued on page 40.)

REFERENCE SECTION

Forth Interest Group

The Forth Interest Group serves both expert and novice members with its network of chapters, *Forth Dimensions*, and conferences that regularly attract participants from around the world. For membership information, or to reserve advertising space, contact the administrative offices:

Forth Interest Group
P.O. Box 8231
San Jose, California 95155
408-277-0668

Board of Directors

Robert Reiling, President *(ret. director)*
Dennis Ruffer, Vice-President
John D. Hall, Treasurer
Wil Baden
Jack Brown
Mike Elola
Robert L. Smith

Founding Directors

William Ragsdale
Kim Harris
Dave Boulton
Dave Kilbridge
John James

In Recognition

Recognition is offered annually to a person who has made an outstanding contribution in support of Forth and the Forth Interest Group. The individual is nominated and selected by previous recipients of the "FIGGY." Each receives an engraved award, and is named on a plaque in the administrative offices.

1979 William Ragsdale
1980 Kim Harris
1981 Dave Kilbridge
1982 Roy Martens
1983 John D. Hall
1984 Robert Reiling
1985 Thea Martin
1986 C.H. Ting
1987 Marlin Ouverson
1988 Dennis Ruffer
1989 Jan Shepherd

ANS Forth

The following members of the ANS X3J14 Forth Standard Committee are available to personally carry your proposals and concerns to the committee. Please feel free to call or write to them directly:

Gary Betts
Unisyn
301 Main, penthouse #2
Longmont, CO 80501
303-924-9193

Mike Nemeth
CSC
10025 Locust St.
Glenndale, MD 20769
301-286-8313

Andrew Kobziar
NCR Medical Systems Group
950 Danby Rd.
Ithaca, NY 14850
607-273-5310

Elizabeth D. Rather
FORTH, Inc.
111 N. Sepulveda Blvd., suite 300
Manhattan Beach, CA 90266
213-372-8493

Charles Keane
Performance Packages, Inc.
515 Fourth Avenue
Watervleit, NY 12189-3703
518-274-4774

George Shaw
Shaw Laboratories
P.O. Box 3471
Hayward, CA 94540-3471
415-276-5953

David C. Petty
Digitel
125 Cambridge Park Dr.
Cambridge, MA 02140-2311

Forth Instruction

Los Angeles—Introductory and intermediate three-day intensive courses in Forth programming are offered monthly by Laboratory Microsystems. These hands-on courses are designed for engineers and programmers who need to become proficient in Forth in the least amount of time. Telephone 213-306-7412.

On-Line Resources

To communicate with these systems, set your modem and communication software to 300/1200/2400 baud with eight bits, no parity, and one stop bit, unless noted otherwise. GEnie requires local echo.

GEnie
For information, call 800-638-9636

- Forth RoundTable
  (ForthNet link)*
  Call GEnie local node, then type M710 or FORTH
  SysOps: Dennis Ruffer (D.RUFFER), Scott Squires (S.W.SQUIRES), Leonard Morgenstern (NMORGENSTERN), Gary Smith (GARY-S)
- MACH2 RoundTable
  Type M450 or MACH2
  Palo Alto Shipping Company
  SysOp: Waymen Askey (D.MILEY)

BIX (ByteNet)
For information, call 800-227-2983

- Forth Conference
  Access BIX via TymeNet, then type j forth
  Type FORTH at the : prompt
  SysOp: Phil Wasson (PWASSON)
- LMI Conference
  Type LMI at the : prompt
  Laboratory MicroSystems products
  Host: Ray Duncan (RDUNCAN)

CompuServe
For information, call 800-848-8990

- Creative Solutions Conference
  Type !Go FORTH
  SysOps: Don Colburn, Zach Zachariah, Ward McFarland, Jon Bryan, Greg Guerin, John Baxter, John Jeppson
- Computer Language Magazine Conference
  Type !Go CLM
  SysOps: Jim Kyle, Jeff Brenton, Chip Rabinowitz, Regina Starr Ridley

Unix BBS's with forth.conf (ForthNet links and reachable via StarLink node 9533 on*

(Reference Section continued)

(Continued from previous page.)

TymNet and PC-Pursuit node casfa on TeleNet.)
- WELL Forth conference
  Access WELL via CompuserveNet
  or 415-332-6106
  Fairwitness: Jack Woehr (jax)
- Wetware Forth conference
  415-753-5265
  Fairwitness: Gary Smith (gars)

PC Board BBS's devoted to Forth (ForthNet links)*
- East Coast Forth Board
  703-442-8695
  StarLink node 2262 on TymNet
  PC-Pursuit node dcwas on TeleNet
  SysOp: Jerry Schifrin
- British Columbia Forth Board
  604-434-5886
  SysOp: Jack Brown
- Real-Time Control Forth Board
  303-278-0364
  StarLink node 2584 on TymNet
  PC-Pursuit node coden on TeleNet
  SysOp: Jack Woehr

Other Forth-specific BBS's
- Laboratory Microsystems, Inc.
  213-306-3530
  StarLink node 9184 on TymNet
  PC-Pursuit node calan on TeleNet
  SysOp: Ray Duncan
- Knowledge-Based Systems
  Supports Fifth
  409-696-7055
- Druma Forth Board
  512-323-2402
  StarLink node 1306 on TymNet
  SysOps: S. Suresh, James Martin, Anne Moore
- Harris Semiconductor Board
  407-729-4949
  StarLink node 9902 on TymNet (toll from Post. St. Lucie)

Non-Forth-specific BBS's with extensive Forth Libraries
- Twit's End (PC Board)
  501-771-0114
  1200-9600 baud
  StarLink node 9858 on TymNet
  SysOp: Tommy Apple
- College Corner (PC Board)
  206-643-0804
  300-2400 baud
  SysOp: Jerry Houston

**ForthNet is a virtual Forth network that links designated message bases in an attempt to provide greater information distribution to the Forth users served. It is provided courtesy of the SysOps of its various links.*

(Continued from page 24.)

considerable function overhead of C, which just is not there in subroutine-threaded Forth. The moral is that you first pick your language and only then do you pick your benchmark!

References

[Bra87] M. Bradley, "Forth to the Future," *Forth Dimensions* (IX/1).

[Cha87] L. Chavez, "A Fast Forth for the 68000," *Dr. Dobb's Journal*, Oct. 1987.

[Joh87] D. Johansen, "Headless Compiler," *Forth Dimensions* (IX/1).

John Redmond is an Associate Professor of Organic Chemistry (Macquarie University, Sydney) with a research interest in the biotechnology of glycoconjugates. His first Forth effort was adapting Loeliger's Z-80 to the Tandy Color Computer. He is a "...sometimes-evenings-when-I-have-time programmer" whose chief disappointment of 1988 consisted of attending a plant pathology conference in Acapulco while Forth's own Charles Moore was visiting Sydney. Mr. Redmond welcomes letters from FD *readers: 23 Mirool Street, West Ryde, NSW 2114, Australia.*

(Continued from page 38.)

produced this very minimal Forth kernel entirely in PostScript lexicon. It is by no means a full system, but does contain the seed elements from which to write a full kernel.

To suggest an interesting on-line guest, leave e-mail posted to GARY-S on GEnie (gars on Wetware and the Well), or mail me a note. I encourage anyone with a message to share to contact me via the above or through the offices of the Forth Interest Group.

(Continued from page 33.)

lems and solutions discussed above; actual address offsets for the various plug-in modifications depend upon the details of the host source. The first two screens define various constants and variables to be used by `HOST`, not to be included in `TARGET`. Screen three sets up the special definition of `INTERPRET` that will compile 0000 when an unfound word is requested.

The host version of `FORTH` has to be renamed to avoid conflict; the substitute name must occupy the same dictionary thread. The (temporary) name for the `TARGET` must have the same number of letters as the host, and must also be in the same thread (screen five). The next two screens are the tricky part: `TARGET` addresses must be substituted in a number of places. Note that these screens do not depend on the dialect being compiled—they are controlled entirely by the `HOST` vocabulary.

Chester H. Page earned his doctorate at Yale and spent some 36 years at the National Bureau of Standards. His first Forth was Washington Apple Pi's fig-FORTH, which he modified to use Apple DOS, then ProDOS, and later to meet the Forth-79 and Forth-83 Standards. Recently, he added many features of F83, including a four-thread dictionary (but no shadow screens).

(Continued from page 30.)

space used by the code is reclaimed. The order in which these actions occur is critical. We must switch back to the standard allocation region before executing the code. Otherwise, words such as `allot` or `,` would effect the compile buffer instead of the standard dictionary. Also, the compiled code must be executed before reclaiming the space it occupies. This allows `]]` and `[[` to be re-entrant, since they could be called via the `execute` in `[[`. For example, in the conditional load, the code in the file may contain interpreted control structures.

A popular Forth programming technique is to temporarily drop out of a colon definition with `[`, do something, then resume compilation with `]`. For consistency, this should also work with interpreted control structures. Thus, the new definitions of `[` and `]` in the listing. `[` saves the `state` in `laststate` and starts interpreting. `]` restores the `state`. A final addition is the initialization of `laststate`:

```
: quit
  ... 0  state  !
  1  laststate  !
  ...
```

In some Forth systems, `]` also contains the compiler loop. Similar modifications should work in such systems, although I haven't tried this.

An unsolved problem with this implementation of interactive control is the increased fragility of the compiler. If something is amiss in the source code being compiled, the compiler can end up stuck in the compile buffer. Since the buffer is small, the space is quickly exhausted.

Interpreted interactive control structures may turn out to be only occasionally useful. On the other hand, as you grow accustomed to them, they may become indispensable. I would be interested to hear about the uses other *Forth Dimensions* readers find for them.

References

[1] Bradley, Mitch. "Interpreting Control Structures—The Right Way," *1987 FORML Conference Proceedings*, pp. 126-130.

John R. Hayes received an M.S. in computer science from Johns Hopkins University in 1986. He has written flight software in Forth for satellite-based magnetometer experiments and for the shuttle-based Hopkins Ultraviolet Telescope.

```
\ Control structures.  This section handles executing control structures
\ interactively.
200 region compileregion               \ a compile buffer
variable compilebuffer                 \ remembers beginning of buffer

: ]]            \ ( --- ) Nest down one control structure level.  If
                \ we were interpreting when ]] was called, switch allocation
                \ to compile buffer.
   state @ 0= if                       \ if we were interpreting
      compileregion allocatefrom       \ switch to compile buffer
      here compilebuffer !             \ remember where we start
      start:                           \ NON-PORTABLE: create code field
   then 1 state +! ;                   \ bump nesting level

: [[            \ ( --- ) Unnest one control structure level.  If we
                \ are now back in interpret state, execute what is in compile
                \ buffer and empty buffer.
   -1 state +!                         \ bump nesting level
   state @ 0= if                       \ if we are now interpreting
      compile exit                     \ compile a return from subroutine
      dp >r  compilebuffer @ dup >r    \ remember where buffer starts
      stdregion allocatefrom           \ revert to standard allocation region
      execute                          \ execute compile buffer
      r> r> !                          \ and empty compile buffer
   then ;

variable laststate                     \ for temporary drop from compile
: [   state @ laststate ! 0 state ! ; immediate
: ]   laststate @ state ! ;
```

FIG CHAPTERS

The FIG Chapters listed below are currently registered as active with regular meetings. If your chapter listing is missing or incorrect, please contact Kent Safford at the FIG office's Chapter Desk. This listing will be updated in each issue of *Forth Dimensions*. If you would like to begin a FIG Chapter in your area, write for a "Chapter Kit and Application." Forth Interest Group, **P.O. Box 8231, San Jose, California 95155**

U.S.A.

- **ALABAMA**
  Huntsville Chapter
  Tom Konantz
  (205) 881-6483

- **ALASKA**
  Kodiak Area Chapter
  Ric Shepard
  Box 1344
  Kodiak, Alaska 99615

- **ARIZONA**
  Phoenix Chapter
  4th Thurs., 7:30 p.m.
  Arizona State Univ.
  Memorial Union, 2nd floor
  Dennis L. Wilson
  (602) 381-1146

- **ARKANSAS**
  Central Arkansas Chapter
  Little Rock
  2nd Sat., 2 p.m. &
  4th Wed., 7 p.m.
  Jungkind Photo, 12th & Main
  Gary Smith (501) 227-7817

- **CALIFORNIA**
  Los Angeles Chapter
  4th Sat., 10 a.m.
  Hawthorne Public Library
  12700 S. Grevillea Ave.
  Phillip Wasson
  (213) 649-1428

  North Bay Chapter
  2nd Sat., 10 a.m. Forth, AI
  12 Noon Tutorial, 1 p.m. Forth
  South Berkeley Public Library
  George Shaw (415) 276-5953

  Orange County Chapter
  4th Wed., 7 p.m.
  Fullerton Savings
  Huntington Beach
  Noshir Jesung (714) 842-3032

  Sacramento Chapter
  4th Wed., 7 p.m.
  1708-59th St., Room A
  Bob Nash
  (916) 487-2044

  San Diego Chapter
  Thursdays, 12 Noon
  Guy Kelly (619) 454-1307

  Silicon Valley Chapter
  4th Sat., 10 a.m.
  H-P Cupertino
  Bob Barr (408) 435-1616

  Stockton Chapter
  Doug Dillon (209) 931-2448

- **COLORADO**
  Denver Chapter
  1st Mon., 7 p.m.
  Clifford King (303) 693-3413

- **CONNECTICUT**
  Central Connecticut Chapter
  Charles Krajewski
  (203) 344-9996

- **FLORIDA**
  Orlando Chapter
  Every other Wed., 8 p.m.
  Herman B. Gibson
  (305) 855-4790

  Southeast Florida Chapter
  Coconut Grove Area
  John Forsberg (305) 252-0108

  Tampa Bay Chapter
  1st Wed., 7:30 p.m.
  Terry McNay (813) 725-1245

- **GEORGIA**
  Atlanta Chapter
  3rd Tues., 7 p.m.
  Emprise Corp., Marietta
  Don Schrader (404) 428-0811

- **ILLINOIS**
  Cache Forth Chapter
  Oak Park
  Clyde W. Phillips, Jr.
  (708) 713-5365

  Central Illinois Chapter
  Champaign
  Robert Illyes (217) 359-6039

- **INDIANA**
  Fort Wayne Chapter
  2nd Tues., 7 p.m.
  I/P Univ. Campus
  B71 Neff Hall
  Blair MacDermid
  (219) 749-2042

- **IOWA**
  Central Iowa FIG Chapter
  1st Tues., 7:30 p.m.
  Iowa State Univ.
  214 Comp. Sci.
  Rodrick Eldridge
  (515) 294-5659

  Fairfield FIG Chapter
  4th Day, 8:15 p.m.
  Gurdy Leete (515) 472-7077

- **MARYLAND**
  MDFIG
  Michael Nemeth
  (301) 262-8140

- **MASSACHUSETTS**
  Boston Chapter
  3rd Wed., 7 p.m.
  Honeywell
  300 Concord, Billerica
  Gary Chanson (617) 527-7206

- **MICHIGAN**
  Detroit/Ann Arbor Area
  Bill Walters
  (313) 731-9660
  (313) 861-6465 (eves.)

- **MINNESOTA**
  MNFIG Chapter
  Minneapolis
  Fred Olson
  (612) 588-9532

- **MISSOURI**
  Kansas City Chapter
  4th Tues., 7 p.m.
  Midwest Research Institute
  MAG Conference Center
  Linus Orth (913) 236-9189

  St. Louis Chapter
  1st Tues., 7 p.m.
  Thornhill Branch Library
  Robert Washam
  91 Weis Drive
  Ellisville, MO 63011

- **NEW JERSEY**
  New Jersey Chapter
  Rutgers Univ., Piscataway
  Nicholas Lordi
  (201) 338-9363

- **NEW MEXICO**
  Albuquerque Chapter
  1st Thurs., 7:30 p.m.
  Physics & Astronomy Bldg.
  Univ. of New Mexico
  Jon Bryan (505) 298-3292

- **NEW YORK**
  Rochester Chapter
  Odd month, 4th Sat., 1 p.m.
  Monroe Comm. College
  Bldg. 7, Rm. 102
  Frank Lanzafame
  (716) 482-3398

- **OHIO**
  Cleveland Chapter
  4th Tues., 7 p.m.
  Chagrin Falls Library
  Gary Bergstrom
  (216) 247-2492

- **Columbus FIG Chapter**
  4th Tues.
  Kal-Kan Foods, Inc.
  5115 Fisher Road
  Terry Webb
  (614) 878-7241

  Dayton Chapter
  2nd Tues. & 4th Wed., 6:30 p.m.
  CFC. 11 W. Monument Ave. #612
  Gary Ganger (513) 849-1483

- **OREGON**
  Willamette Valley Chapter
  4th Tues., 7 p.m.
  Linn-Benton Comm. College
  Pann McCuaig (503) 752-5113

- **PENNSYLVANIA**
  Villanova Univ. Chapter
  1st Mon., 7:30 p.m.
  Villanova University
  Dennis Clark
  (215) 860-0700

- **TENNESSEE**
  East Tennessee Chapter
  Oak Ridge
  3rd Wed., 7 p.m.
  Sci. Appl. Int'l. Corp., 8th Fl.
  800 Oak Ridge Turnpike
  Richard Secrist
  (615) 483-7242

- **TEXAS**
  Austin Chapter
  Matt Lawrence
  PO Box 180409
  Austin, TX 78718

  Dallas Chapter
  4th Thurs., 7:30 p.m.
  Texas Instruments
  13500 N. Central Expwy.
  Semiconductor Cafeteria
  Conference Room A
  Clif Penn (214) 995-2361

  Houston Chapter
  3rd Mon., 7:30 p.m.
  Houston Area League of PC Users
  1200 Post Oak Rd.
  (Galleria area)
  Russell Harris
  (713) 461-1618

- **VERMONT**
  Vermont Chapter
  Vergennes
  3rd Mon., 7:30 p.m.
  Vergennes Union High School
  RM 210, Monkton Rd.
  Hal Clark (802) 453-4442

- **VIRGINIA**
  First Forth of Hampton Roads
  William Edmonds
  (804) 898-4099

  Potomac FIG
  D.C. & Northern Virginia
  1st Tues.
  Lee Recreation Center
  5722 Lee Hwy., Arlington
  Joseph Brown
  (703) 471-4409
  E. Coast Forth Board
  (703) 442-8695

  Richmond Forth Group
  2nd Wed., 7 p.m.
  154 Business School
  Univ. of Richmond
  Donald A. Full
  (804) 739-3623

- **WISCONSIN**
  Lake Superior Chapter
  2nd Fri., 7:30 p.m.
  1219 N. 21st St., Superior
  Allen Anway (715) 394-4061

INTERNATIONAL

- **AUSTRALIA**
  Melbourne Chapter
  1st Fri., 8 p.m.
  Lance Collins
  65 Martin Road
  Glen Iris, Victoria 3146
  03/29-2600
  BBS: 61 3 299 1787

  Sydney Chapter
  2nd Fri., 7 p.m.
  John Goodsell Bldg., RM LG19
  Univ. of New South Wales
  Peter Tregeagle
  10 Binda Rd.
  Yowie Bay 2228
  02/524-7490
  Usenet
  tedr@usage.csd.unsw.oz

- **BELGIUM**
  Belgium Chapter
  4th Wed., 8 p.m.
  Luk Van Loock
  Lariksdreff 20
  2120 Schoten
  03/658-6343

  Southern Belgium Chapter
  Jean-Marc Bertinchamps
  Rue N. Monnom, 2
  B-6290 Nalinnes
  071/213858

- **CANADA**
  BC FIG
  1st Thurs., 7:30 p.m.
  BCIT, 3700 Willingdon Ave.
  BBY, Rm. 1A-324
  Jack W. Brown
  (604) 596-9764
  BBS (604) 434-5886

  Northern Alberta Chapter
  4th Sat., 10a.m.-noon
  N. Alta. Inst. of Tech.
  Tony Van Muyden
  (403) 486-6666 (days)
  (403) 962-2203 (eves.)

  Southern Ontario Chapter
  Quarterly, 1st Sat., Mar., Jun., Sep., Dec., 2 p.m.
  Genl. Sci. Bldg., RM 212
  McMaster University
  Dr. N. Solntseff
  (416) 525-9140 x3443

- **ENGLAND**
  Forth Interest Group-UK
  London
  1st Thurs., 7 p.m.
  Polytechnic of South Bank
  RM 408
  Borough Rd.
  D.J. Neale
  58 Woodland Way
  Morden, Surry SM4 4DS

- **FINLAND**
  FinFIG
  Janne Kotiranta
  Arkkitehdinkatu 38 c 39
  33720 Tampere
  +358-31-184246

- **HOLLAND**
  Holland Chapter
  Vic Van de Zande
  Finmark 7
  3831 JE Leusden

- **ITALY**
  FIG Italia
  Marco Tausel
  Via Gerolamo Forni 48
  20161 Milano
  02/435249

- **JAPAN**
  Japan Chapter
  Toshi Inoue
  Dept. of Mineral Dev. Eng.
  University of Tokyo
  7-3-1 Hongo, Bunkyo 113
  812-2111 x7073

- **NORWAY**
  Bergen Chapter
  Kjell Birger Faeraas,
  47-518-7784

- **REPUBLIC OF CHINA**
  R.O.C. Chapter
  Chin-Fu Liu
  5F, #10, Alley 5, Lane 107
  Fu-Hsin S. Rd. Sec. 1
  TaiPei, Taiwan 10639

- **SWEDEN**
  SweFIG
  Per Alm
  46/8-929631

- **SWITZERLAND**
  Swiss Chapter
  Max Hugelshofer
  Industrieberatung
  Ziberstrasse 6
  8152 Opfikon
  01 810 9289

- **WEST GERMANY**
  German FIG Chapter
  Heinz Schnitter
  Forth-Gesellschaft C.V.
  Postfach 1110
  D-8044 Unterschleissheim
  (49) (89) 317 3784
  Munich Forth Box:
  (49) (89) 725 9625 (telcom)

SPECIAL GROUPS

- **NC4000 Users Group**
  John Carpenter
  1698 Villa St.
  Mountain View, CA 94041
  (415) 960-1256 (eves.)

CALL FOR PAPERS

for the twelfth annual

FORML CONFERENCE

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

Following Thanksgiving, November 23–25, 1990

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California U.S.A.

Conference Theme: Forth in Industry

Papers are invited that address relevant issues in the development and use of Forth in industry. Papers about other Forth topics are also welcome.

Mail abstract(s) of approximately 100 words by September 1, 1990 to FORML, P.O. Box 8231, San Jose, CA 95155.

Completed papers are due November 1, 1990.

Registration information may be obtained by telephone from the Forth Interest Group business office (408) 277-0668 or write to FORML, P.O. Box 8231, San Jose, CA 95155.

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.

Forth Interest Group
P.O.Box 8231
San Jose, CA 95155

Second Class
Postage Paid at
San Jose, CA