$10 Volume XIII, Number 6 March 1992 April

FORTH DIMENSIONS

Curly Control Structure Set (I)

Forth Systems Comparison

PCYerk Classes

Minimal Forth Wordsets

Contents

Editorial

New Conte$t for Forth Authors!

With this issue, many of you will be due to renew your membership in the Forth Interest Group. This is a year to do so promptly and to make a gift membership or two for the office, a co-worker, or friend—here are a few things to look for in coming issues of *Forth Dimensions*:

A West-coast group of Forth adepts is producing a series of articles applying Forth to hands-on, hardware-software projects that you can do—a laboratory for increasing your Forth proficiency at the workbench. Vendors and developers will have more opportunities to contribute technical and industry information in ways that will show what they are doing successfully and where Forth is excelling in real-world application. And we have scheduled tutorials about traditional tools like `CREATE DOES>` as well as the control structures that will be introduced in ANS Forth. More than ever we believe that, from beginner to expert, every Forth user and project manager will want to receive the vital information that will be appearing here.

Reader participation has always been a key element of this publication. Your contributions are the lifeblood of our pages, dramatically helping to chart our direction. We not only welcome your own articles and letters to the editor, we need them.

* * *

FD can now announce the third in a series of contests for Forth authors. The first called for entries about Forth hardware, and the winners were published in our issue XI/6. More recently, the winners of our object-oriented Forth contest appeared in issue XII/5. Drawing on feedback from Forth vendors, the theme of our current contest is Forth in large-scale applications.

This is our first call for papers about "FORTH ON A GRAND SCALE." This theme applies equally to projects requiring multiple programmers, and to applications or systems consisting of large amounts of code and/or of significant complexity. Papers will be refereed. To encourage entries, the author of the winning article will receive $500, the second-place $250, and the third-place $100. Articles will be evaluated for publication even if they do not win a cash prize.

You need not have been personally involved in the subject of your entry, just write about it in sufficient technical detail, and address the particular challenges that were faced and describe how (or whether!) they were overcome. Chances are, if you think a subject *might* fit the theme of this contest, the judges will be anxious to include it in their evaluations—so get started soon. *The deadline for contest entries is August 3, 1992.* Mail a hard copy and a diskette (Macintosh 800K or PC preferred) to the Forth Interest Group, P.O. Box 8231, San Jose, California 95155; or mail the hard copy and upload an ASCII version to MARLIN.O on GEnie's e-mail service with an attached note describing the file and compression/ archive format, if any. We all look forward to receiving your contribution!

* * *

At the other end of the scale we have minimal Forths. How small can you get and still have a language? What are the fewest required words in Forth? That is the on-line discussion excerpted in "The Best of GEnie" this month. Elsewhere in this issue, you will find supplemental code to the object-oriented Forth "PCYerk" by Rick Grehan, and a meaty discussion of control structures by Kourtis Giorgio that will be concluded in the next issue. Finally, Guy Kelly shares his FORML paper with *FD* readers. It is a significant piece of work that shows what goes into evaluating Forth systems, and we thank him for allowing us to publish it here. It demonstrates the difficulty of doing head-to-head product comparisons, and is the first substantial attempt we know of to do so thoroughly and objectively. Pay special heed to his warning that benchmark excellence alone does not mean that any single system will be the right one every purpose!

—*Marlin Ouverson*
Editor

Forth Dimensions
Volume XIII, Number 6
March-April 1992

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Anna Brereton

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

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $40 per year ($52 overseas air); student rate available. 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. Fax: 408-286-8988

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

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $40/46/52 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 8231, San Jose, CA 95155."

Forth Systems Comparisons

Guy M. Kelly
La Jolla, California

Code fragments and benchmarks for several of the Forths for the PC are outlined to illustrate various tradeoffs and their effect on performance.

The following list represents some of the Forths I have been able to study. They span a wide range of implementation tradeoffs and provide some insight into the results of these tradeoffs.

Forth	Model	Author(s)	Status
BBL	83	Green	public
eForth	X3J14	Muench & Ting	public
F83	83	Laxen & Perry	public
F-PC	83	Zimmer & Smith	public
Fifth		Click & Snow	share
HS/FORTH	*	Callahan	commercial
KForth	experimental	Kelly	copyrighted
LaFORTH	experimental	Smith & Stuart	copyrighted
MMSFORTH	79	Miller et al.	commercial
MVP-FORTH	79	Haydon	public
PC-Forth	83	Kelly	public
PMFORTH	fig	Moreton	commercial
polyFORTH	83	Moore et al.	commercial
Pygmy	cmFORTH	Sergeant	copyrighted
riFORTH	cmFORTH	Illyes	copyrighted
UniForth	83	Hendon?	share
Upper Deck	83	Graves	commercial
UR/FORTH	83	Duncan & Wilton	commercial
ZEN	X3J14	Tracy	copyrighted

*Includes overlays to convert to fig, 79, or 83 standard.

Some of these Forths are available in different packages including public, share, or commercial versions. The version tested had the status indicated. The non-commercial versions are typically available at no charge, the commercial versions are typically copyrighted. The model does not imply compatibility.

These Forths cover a range of categories and complexities, as Table One illustrates.

The author presented this paper at the 1991 FORML Conference. Those who were unable to attend that event can order the complete proceedings from the Forth Interest Group.

Segment Models

Assuming four logical segments (not including the stacks), there are 15 different models. The following lists these models and indicates their use by each of the Forths studied.

C+L+D+H	e, F83, La, MMS, MVP, PC, pygmy, ri, Uni
C+L+H D	polyFORTH
C+L D+H	PM, ZEN
C L+D H	BBL, HS/FORTH, UR/FORTH
C+D L H	F-PC
C L D H	KForth, Upper Deck

Not found:
C L+D+H
L C+D+H
H C+L+D
C+D L+H
C+H L+D
C L D+H
C D L+H
L D C+H
D H C+L

Descriptions

Brief descriptions of most of the Forths tested are included at the end of this paper (all assembly code is in a common format).

Benchmarks

While studying the various threading, stack, and segmenting methods it seemed that a set of simple benchmarks could help in evaluating the performance trade-offs. The benchmarks arrived at are specifically aimed at the attributes studied and do not necessarily correlate with real applications.

Threading

There are two aspects of threading in Forth to be evaluated. The efficiency of incrementing the Forth instruction-pointer and the efficiency of nesting (and unnesting).

The following threading benchmarks were used:

```
\ Empty loop: Empty = XX
: X       ( -- ) 30,000 0 DO      LOOP ;
: XX      ( -- )        5 0 DO  X  LOOP ;

\ Threading: Thread = YY - XX
CODE NC   ( -- ) NEXT,  END-CODE
: Y       ( -- )
  30,000 0 DO  NC NC NC NC NC NC  LOOP ;
: YY      ( -- )        5 0 DO  Y  LOOP ;

\ Nesting1: Nest1 = ZZ - XX
: N:      ( -- ) ;
: Z       ( -- )
  30,000 0 DO  N: N: N: N: N: N:  LOOP ;
: ZZ      ( -- )        5 0 DO  Z  LOOP ;

\ Nesting2: Nest2 = WW - XX
: W1 ;  : W2 W1 ;  : W3 W2 ;  : W4 W3 ;
: W5 W4 ;  : W6 W5 ;
: W       ( -- ) 30,000 0 DO  W6 LOOP ;
: WW      ( -- )        5 0 DO  W  LOOP ;
```

The two nesting benchmarks should be equivalent but can be very different depending upon any optimization applied.

Top-of-Stack Location

```
\ Primitives: Prims = QQ - XX
\ Exercise:  variable constant  @  !  + DUP
\            SWAP  OVER  DROP

   VARIABLE LOC
10 CONSTANT TEN

: NULL    ( -- )
  TEN DUP  LOC SWAP  OVER !  @ +  DROP ;
: Q       ( -- ) 30,000 0 DO  NULL  LOOP ;
: QQ      ( -- )        5 0 DO  Q  LOOP ;
```

Other Benchmarks

To satisfy the curious, the "standard" Sieve benchmark and a simple interpreting-time benchmark are included.

```
\ Sieve: Sieve = 10 0 do DO-PRIME loop

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
    THEN
  LOOP  . ;
```

Table One.

Forth	**Threading**	**Width**[1]	**Stack**	**Segments**[2]
BBL	direct	32/16os	in reg	N=C,m(L+D),n*H,S+B
eForth	direct	16/16	on stack	1 (sep. heads)
F83	indirect	16/16	on stack	1
F-PC	direct	16/16pp	on stack	3=C+D+S,mL,H
Fifth	subroutine?	32/?	?	?
HS/FORTH	indirect	16/16	in reg	N=n*C,n*(L+D),n*H,S
KForth	direct	16/16	in reg	5=C,L,D,H,S
LaFORTH	direct	16/16	in reg	2(2nd for text files)
MMSFORTH	indirect	16/16	on stack	1(non-DOS), 2=C+L+D,H
MVP-FORTH	indirect	16/16	on stack	1
PC-Forth	indirect	16/16	on stack	1
PMFORTH	direct	16/16	on stack	2=C+L,D+H+S
polyFORTH	indirect	16/16	on stack	N=n*(C+L+H),(D+S),n*D
Pygmy	direct	16/16	in reg	1
riFORTH	subroutine	16/16	in reg	1
UniForth	indirect	16/16	on stack	1
Upper Deck	direct	16/16	in reg	5=C,L,D,H,S
UR/FORTH	direct	16/16	in reg	4=C,L+D,H,S
ZEN	direct	16/16	in reg	2=C+L,D+H+S

1. Width given as: stack-width/token-width; os indicates token is an offset into the code segment, pp indicates token is a 16-bit paragraph address.
2. Code, List, Data, Head, and Stack; m(L) indicates one meg. of paragraph space for tokens; n*(L+D) or n*H indicates n 64K segments for lists+data or heads.

Interpret-time benchmark.

```
\ Interpret-time: Loads (tests: WORD, NUMBER, and FIND etc.)

        99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP 99 DROP
```

Note: `Loads` is strongly influenced by the search method and, in many Forths, by the number of words in the dictionary.

Results

Initial testing was done on a 20 MHz '386; however, if the code or list addresses were moved from non-word to word boundaries, the times were significantly improved. This effect was noticed using PC-Forth and only investigated for PC-Forth and PC/FORTH (a now discontinued product from LMI, which resisted attempts to force code or list addresses to non-word boundaries). A 4.77 MHz 8088 did not exhibit this behavior and was used to obtain the results listed in Tables Two and Three.

The benchmarks arrived at are specifically aimed at the attributes studied and do not necessarily correlate with real applications.

Table Two.

Forth	Type	Empty	Thread	Nest1	Nest2	Prims	Sieve	Loads
BBL	D-R-L	4.0	5.9	41.4	41.4	33.1	49.0	4.3
eForth	D-S	4.3	---	42.2	---	---	---	---
F83	I-S-J	5.4	15.1	43.4	43.4	41.5	68.1	3.8
F-PC	D-S-P	3.1	11.0	46.4	46.4	30.3	44.9	0.9
Fifth	S-?	7.1	---	20.9	20.9	70.4	97.2	---
HS/FORTH	I-R	5.5	9.8	34.2	33.8	28.5	48.8	0.7
KForth	D-R	2.9	5.8	25.8	25.8	21.6	36.2	21.9
LaFORTH	D-R	3.6	5.8	29.9	29.7	26.9	36.3	0.5
MMSFORTH	I-S	5.2	10.0	37.4	37.4	33.4	55.6	0.5
MVP-FORTH	I-S-J	14.2	19.8	53.3	53.3	50.8	---	8.5
PC-Forth	I-S	3.6	10.0	34.5	34.5	32.7	54.8	8.4
PMFORTH	D-S	5.5	9.6	43.5	43.5	39.8	70.?	15.1
polyFORTH	I-S	4.8	9.9	34.5	34.5	31.5	52.9	1.0
Pygmy	D-R	3.2	5.9	32.0	32.0	23.5	39.7	4.7
riFORTH	S-R	13.1	9.6	9.6	5.4	9.6	34.8	6.8
UniForth	I-S	3.6	10.7	35.9	35.8	33.3	---	2.7
Upper Deck	D-R	4.2	5.9	32.1	32.1	23.8	39.8	0.5
UR/FORTH	D-R	2.1	5.8	31.9	32.0	24.3	38.2	0.5
ZEN	D-R	3.6	6.8	34.1	33.8	27.4	44.6	4.4

All times in seconds, all measurements on a 4.77 MHz 8088 PC.

Table Three.

Forth	Type[1]	Empty	Thread	Nest1	Nest2	Prims	Sieve	Loads
riFORTH	S-R	13.1	9.6	9.6	5.4	9.6	34.8	6.8
Fifth	S-?-?	7.1	---	20.9	20.9	70.4	97.2	---
KForth	D-R	3.0	5.8	25.8	25.8	21.6	36.2	21.9
LaFORTH	D-R	3.6	5.8	29.9	29.7	26.9	36.3	0.5
UR/FORTH	D-R	3.0	5.8	31.9	32.0	24.3	38.2	0.5
Pygmy	D-R	3.2	5.9	32.0	32.0	23.5	39.7	4.7
Upper Deck	D-R	4.2	5.9	32.1	32.1	23.8	39.8	0.5
ZEN	D-R	3.6	6.8	34.1	33.8	27.4	4.6	4.4
HS/FORTH	I-R	5.5	9.8	34.2	33.8	28.5	48.8	0.7
polyFORTH	I-S	4.8	9.9	34.5	34.5	31.5	52.9	1.0
PC-Forth	I-S	3.6	10.0	34.5	34.5	32.7	54.8	8.4
UniForth	I-S	3.6	10.7	35.9	35.8	33.3	---	2.7
MMSFORTH	I-S	5.2	10.0	37.4	37.4	33.4	55.6	0.5
BBL	D-R-L	4.0	5.9	41.4	41.4	33.1	49.0	4.3
eForth	D-S	4.3	---	42.2	---	---	---	---
F83	I-S-J	5.4	15.1	43.4	43.4	41.5	68.1	3.8
PMFORTH	D-S	5.5	9.6	43.5	43.5	39.8	70	15.1
F-PC	D-S-P	3.1	11.0	46.4	46.4	30.3	44.9	0.9
MVP-FORTH	I-S-J	14.2	19.8	53.3	53.3	50.8	---	8.5

Sorted by nesting time.

1. Type: Indirect, Direct, or Subroutine threading, stack-top in Register, or on Stack, 1-meg. Lists, 1-meg. lists on Paragraphs, Jump to NEXT.

Timing

In general the results were as follows (fastest to slowest):

- subroutine threading; top-of-stack in register,
- direct threading; top-of-stack in register,
- indirect threading; top-of-stack in register,
- indirect threading; top-of-stack in memory,
- direct threading; top-of-stack in memory.[1]

1. Expected to be third, not last (PMFORTH was the only example).

To obtain the maximum advantage from Forth, one should understand the rationale for its structure and its inherent strengths and weaknesses.

"16-bit"	**Threading**	**Nesting**	**Primitives**	**Sieve**
Subroutine	10	5/10	10	35
Direct	6-07	25-34	22-27	36-45
Indirect	10	34-37	28-33	48-56
I(JMP NEXT)	15-20	43-53	42-50	68
"16-bit paragraphs"				
F-PC (Dir)	11	46	30	45
"32-bit"				
BBL (Dir)	06	41	33	49
Fifth (Sub)	--	21	70	97

Optimization

Two of the Forths allowed optimization of user specified words. The results obtained using the optimizers are shown in Table Four.

Comments

Several aspects of these Forths make direct comparison difficult. Most of them do not automatically optimize their code nor do they directly span multiple segments. However, riFORTH does automatic optimization; polyFORTH has multiple C+L+H spaces; BBL, F-PC, and Fifth have up to one meg. of list space; F83 and MVP-FORTH have a central `NEXT`; MVP-FORTH and PMFORTH have inefficient versions of `NEXT`; KForth does high-level parsing; eForth interprets files via a serial link; and LaFORTH uses a 64K text buffer.

Further Tests

Because of the differences mentioned above, a test-set of five different versions of Forth were produced. They were all derived from riFORTH (a subroutine-threaded Forth available in a minimum number of screens). The versions (including riFORTH) were:

Name	Model		
	threading	top-of-stack	segmentation
S-R-S	Subroutine	in Register	Single (riFORTH)
D-R-M	Direct	in Register	Multiple
D-S-M	Direct	on Stack	Multiple
I-R-M	Indirect	in Register	Multiple
I-S-M	Indirect	on Stack	Multiple
D-R-S	Direct	in Register	Single

The versions were optimized for speed at the expense of size. All models used an in-line `NEXT` and in-line nest, `LIT`, etc., where possible. The benchmark results (sorted by nesting time) are given in Table Five.

Note that for riFORTH, Nest2 is almost twice as fast as Nest1 while Thread and Nest1 take the same time. This is because riFORTH is subroutine-threaded and has built-in optimization. Referring to the nesting benchmarks, the "code" no-op `NC`, and the "colon" no-ops `:N` and `W1` all compile as return instructions. However, `W2` is compiled as a jump to `W1`, `W3` as a jump to `W2`, etc., thus doing five jumps and a return inside the `W` loop instead of six call-return pairs. Also note that the Prims are executed much faster for riFORTH than the other versions (because riFORTH drops adjacent `XCHG BP,SI` pairs from "code macros" as it compiles them into the list field of a colon definition), while the Sieve (which uses a high-level `DO LOOP`) is only slightly faster.

The apparent anomaly among the other versions is D-R-S, the only one of the five that is not multi-segment. It nests more slowly but does Prims and Sieve faster than D-S-M because nest, `LIT`, and `VARIABLE` cannot be as highly optimized for speed.

Table Four. Results using optimizers.

Forth	Type	Empty	Thread	Nest1	Nest2	Prims	Sieve	Loads
HS/FORTH	I-R	5.5	9.8	34.2	33.8	28.5	48.4	0.7
optimized		3.2	0.0	0.0	0.0	5.2	12.9	0.7
UR/FORTH	D-R	3.0	5.8	31.9	32.0	24.3	38.2	0.5
optimized		0.8	12.9	23.1	23.1	6.3	7.5	0.5

Table Five. Performance of test-set versions of Forth.

Name	Empty	Thread	Nest1	Nest2	Prims	Sieve
S-R-S	13.1	9.6	9.6	5.4	9.6	34.8
D-R-M	2.9	5.8	25.8	25.8	21.5	36.2
D-S-M	2.9	5.8	25.8	25.8	25.1	42.3
D-R-S	2.9	5.8	31.9	32.3	23.4	37.9
I-R-M	3.6	10.0	33.8	33.5	28.8	47.7
I-S-M	3.6	10.0	33.8	33.5	32.1	53.2

The following lists the versions of NEXT, nest, EXIT, literal, CONSTANT, VARIABLE, @, !, and + used in these models.

Code Fragments

	D-R-M	D-S-M	I-R-M	I-S-M	D-R-S
EXIT	MOV SI,[BP] INC BP INC BP NEXT	see D-R-M <--	see D-R-M <--	see D-R-M <--	see D-R-M <--

nest	---
	DEC BP
	DEC BP
	MOV [BP],SI
	MOV SI,addr
	NEXT
		see D-R-M
		<--
			see D-R-M
			<--
				see D-R-M
				<--
					JMP nest
					DEC BP
					DEC BP
					MOV[BP],SI
					ADD AX,3
					MOV SI,AX
					NEXT

NEXT	LODSW JMP AX	see D-R-M <--	LODSW XCHG AX,DX JMP [DX]	LODSW XCHG AX,BX JMP [BX]	see D-R-M (LODSW JMP AX)
CON	PUSH BX MOV BX,# NEXT	see D-R-M <--	see D-R-M <--	see D-R-M <--	see D-R-M <--
VAR	see CON ^	see CON ^	see CON ^	see CON ^	ADD AX,3 PUSH BX XCHG AX,BX NEXT
LIT	see CON ^	MOV BX,# PUSH BX NEXT	LODSW PUSH BX XCHG AX,BX NEXT	LODSW PUSH AX NEXT	LODSW PUSH BX XCHG AX,BX NEXT
@	ES: MOV BX,[BX] NEXT	POP BX ES: PUSH [BX] NEXT	ES: MOV BX,[BX] NEXT	POP BX ES: PUSH [BX] NEXT	ES: MOV BX,[BX NEXT
!	ES: POP [BX] POP BX NEXT	POP BX ES: POP [BX] NEXT	ES: POP [BX] POP BX NEXT	POP BX ES: POP [BX] NEXT	ES: POP [BX] POP BX NEXT
+	POP AX ADD BX,AX NEXT	POP BX POP AX ADD BX,AX PUSH BX NEXT	POP AX ADD BX,AX NEXT	POP BX POP AX ADD BX,AX PUSH BX NEXT	POP AX ADD BX,AX NEXT

Observations

Ignoring the various anomalies, the spread in performance among the Forths for most benchmarks is about a factor of two (about a factor of 1.5 among the test-set versions.) This seems a small gain considering both the efforts that have gone into the various implementations and the resulting lack of internal consistency from one implementation to the next. (It was, however, easier to handle these inconsistencies when writing the various versions of the benchmarks than to handle the inconsistencies among different assemblers supplied with the various Forths.)

Specifics

The data for the D-R-M, D-S-M, I-R-M, and I-S-M versions yields the following ratios:

Indirect/Direct	= 1.7:1 (ratio of times for Thread)
Indirect/Direct	= 1.3:1 (ratio of times for Nest1 or Nest2)
Indirect/Direct	= 1.3:1 (ratio of times for Prims or Sieve)
Stack/Reg (Dir)	= 1.2:1 (ratio of times for Prims or Sieve)
Stack/Reg (Ind)	= 1.1:1 (ratio of times for Prims or Sieve)
I-S-M/D-R-M	= 1.5:1 (ratio of times for Prims or Sieve)
D-R-S/D-R-M	= 1.1:1 (ratio of times for Prims or Sieve)

These ratios indicate that changing from indirect-threading to direct-threading in the multi-segment version provides about a 30% speed-up, while changing from top-of-stack on the stack to top-of-stack in a register provides about a 10% speed-up. Changing both provides about a 50% speed-up.

The segment model affects performance in the case shown above by about 5–10% because the D-R-S version does not permit the best possible optimization of the Forth virtual machine for speed (as shown by the code fragments on the preceding page).

A more significant reason for segmentation is that it provides separation of the components of a Forth word and can provide more memory in which to program. For example, separating the headers from the rest of the words can provide more program space or can make an application smaller and much harder to disassemble.

Another reason for segmentation is that more and more operating systems restrict the use of data and code in the same memory "hunk." These systems normally restrict read-write access to data structures in the code hunk, making an application either use separate hunks for code and data or use the operating system to overcome such restrictions (with possible performance penalties).

Opinions

Selecting one Forth over another for a typical gain of 50% in performance may be the wrong reason to make the choice. Changing from an 8088 to a faster member of the family, changing an algorithm, or using the optimizers available with several of the Forths can result in gains of from three to more than 30.

The following considerations would seem at least as important:

- quality and completeness of the implementation,
- availability and appropriateness of additional modules,
- availability and quality of support including documentation,
- transportability of source and ease of use,
- application-size supported.

Notice that price is not in the above list. If you are going to use the Forth for a commercial application, even the highest priced commercial Forth is inexpensive if it has features that are important to your application and will allow you to finish your project significantly faster than you otherwise would.

A particular consideration these days is the size of the application supported. Most commercial applications are big and growing bigger, especially those that have to run under most of the current graphical user interfaces. The typical single-segment Forth, even with overlays, is hard pressed to support the bloated programs that seem to be required. (Even embedded systems are getting larger, although minimizing their size is still very important.)

Most of the Forths reviewed do not easily support large programs and among those that do, there are a variety of trade-offs that need to be considered. Some of the Forths that seem to support large programs have limitations on the space available for code and/or data, others do not. Some require significantly more memory for a given application than others. The segmentation information and the code-fragments presented for the Forths provide some insight as to the advantages and limitations of the various Forths.

Another consideration that is becoming more important, at least in the PC world, is the ease with which foreign libraries and facilities (DLLs, OLE, etc.) can be accommodated. Most of the Forths reviewed have no built-in capability, a few do. If this is an important consideration, one should investigate the support for interfacing to other programs and libraries that may be available.

Most of the Forths reviewed claim to support multitasking. If this is an important feature, be warned that the support provided is usually minimal. Further, almost none of these Forths provide useful multiuser support.

Forth

For those wishing to evaluate Forth, important considerations include ease of use (including DOS interface and available editors), standardization, and adherence to available Forth texts.

Another consideration that is important when considering a Forth is whether you are going to approach it as a black box, or whether you are interested in understanding its internal structure. To obtain the maximum advantage from Forth, one should understand both the rational for its structure and its inherent strengths and weaknesses. This requires at least some understanding of the internals of the version being used and becomes more important as an application becomes more complex. The Forths reviewed range from simple to very complex and the documentation of their structure ranges from nonexistent to well detailed.

Further, some provide complete source code and some do not (although you can usually obtain it for a fee). At the most advanced level, those that supply source provide it either as native Forth code with a metacompiler or as assembly code for use with a standard assembler. Be warned, most metacompilers are difficult to master at best and you usually require some understanding of them to follow the accompanying Forth source.

Finally, remember... Forth can never (well, hardly ever) be too small or too fast—especially for all those big and slow applications.

Forth Assemblers (an aside)

How to move the contents of memory (pointed to by the contents of register BX) into a register (AX in this case):

`MOV AX, [BX]`	opcode destination source
`AX, [BX] MOV`	destination source opcode
`AX [BX] MOV.`	same order, trailing period
`[BX] AX MOV`	source destination opcode
`[BX] AX MOV,`	same order, trailing comma
`[BX] AX LDA,`	same order, different opcode
`3) 0 MOV,`	same order, different register "names"
`3) 0 LDA`	same order, different opcode

and there are probably more (and you haven't seen how to index yet!).

The code fragments are all given in a standard format. This does not reflect the flavor of the assembler mnemonics of the various Forths studied (as hinted at above) but does make it easier to understand and compare the examples.

How to open a file and load a program in the various Forths.

Forth	**Case**	**Method**

BBL	screen-file	`CACHE-NAME 30 EXPECT <cr> BBLBENCH <cr>`
		`0 CACHE-NAME 8 + ! <cr>`
		`OPEN-CACHE 1 LOAD <cr>`
		(my old version did not have `USING`)

eForth	text-lines	via serial channel
F83	screen-file	`OPEN F83BENCH. 1 LOAD <cr>`
F-PC	text-file	`FLOAD BENCH <cr>`
Fifth	text-file	`L SIEVE.FIV <cr> C <cr>`
HS/FORTH	text-file	`FLOAD HSFBENCH <cr>`

LaForth	text-file	`LA LA.HI <cr> (from DOS) LT RUN <cr>`
		`BT MT TEXT LABENCH^Z BT OPEN . (handle)`
		`BT size handle READ TP +! <cr>`
		`0 LT DROP TP @ XC! LT RUN <cr>`
		(couldn't find a better way - must be one?)

MMSFORTH	screens	`400 LOAD <cr>` (non-DOS, see MMSBENCH)
MVP-FORTH	screens	`342 (or 171) LOAD <cr>` (see MVPBENCH)
PC-Forth	screen-file text-file	`INCLUDE PCBENCH <cr>` or `INCLUDE PCBNECH <cr>`
PMFORTH	screen-file (in PMfile)	`OPEN B:PBENCH <cr>` `1 SFLOAD <cr>`
polyFORTH	screen-file	`CHART PCBENCH 1201 LOAD <cr>` or `1 LOADUSING PCBENCH <cr>`

Pygmy	screen-file	`NAMEZ: PYGBENCH <cr>`
		`600 PYGBENCH 2 UNIT <cr>`
		`2 OPEN 1 LOAD <cr>`

riFORTH	screen-file	`RIFORTH RIBENCH <cr>` (from DOS) `2 LOAD <cr>` (screens start from 1)
UniForth	screen-file	`UNIFORTH UNIBENCH <cr>` (from DOS) `1 LOAD <cr>`
Upper Deck	text-file	`CAPS ON RELOAD BENCH <cr>`
UR/FORTH	screen-file	`ASM USING LMIBENCH. 1 LOAD <cr>`
ZEN	text-file	`INCLUDE ZENBENCH. <cr>`

BBL

V1? 10/25/86, Green
public[1]

Written by Roedy Green to use as the tool for rewriting Abundance (a vast database program and application). Source code for BBL is in assembler.

A direct-threaded 32-bit implementation with the top-of-stack in a register. A multiple-segment model which interprets from screen files.

Segments (max. size each):

Code(64K) Lists+data(1 meg.) Heads(n*64K)
Stack+Block(64K)

Register use:

```
AX = W            SI = IP         CS = code
BX = tos(lsw)     DI = 0 (lit)    DS = lists+data (seg/off of )
CX = tos(msw)     BP = RP         ES = lists+data (32 bit addr)
DX = -            SP = SP         SS = stacks
```

```
Next            Nest (bigger, faster)     Unnest
LODSW           XCHG SP,BP                XCHG SP,BP
JMP AX          PUSH SI                   POP DS
                PUSH DS                   POP SI
                XCHG SP,BP                XCHG SP,BP
                MOV DX,xxx_pfa(seg)       NEXT
                MOV SI,xxx_pfa(offset)
                MOV DS,DX
                NEXT
```

```
@                   !                   Nest (slower, smaller)
MOV ES,CX           MOV ES,CX           MOV DS,xxx_pfa(seg)
MOV CX,ES:[BX+2]    POP ES:[BX+2]       MOV AX,xxx_pfa(off)
MOV BX,ES:[BX]      POP ES:[BX]         JMP DOCOL
NEXT                POP CX
                    POP BX       DOCOL: XCHG SP,BP
                    NEXT                PUSH SI
                                        PUSH DS
                                        XCHG SP,BP
                                        MOV DS,DX
                                        MOV SI,AX
                                        NEXT
```

```
Constant (in-line)  Variable              +
PUSH BX             PUSH BX               POP DX
PUSH CX             PUSH CX               POP AX
MOV BX,#(lsw)       MOV CX,xxx_pfa(seg)   ADD BX,AX
MOV CX,#(msw)       MOV BX,xxx_pfa(off)   ADC CX,DX
NEXT                NEXT                  NEXT
```

Notes: compiled tokens are offsets into the code segment.

1. Not for military use.

---

eForth

V1.0 7/27/90, Muench et al.
public

eForth has been proposed as the successor to fig-FORTH for porting to current microprocessors, is available in several implementations, and is tailored toward transportability, ROMmability, and use in embedded controllers. Source code is usually in assembler.

A direct-threaded 16-bit implementation with the top-of-stack on the stack. It has separated heads in a single common segment and usually interprets source code from a host serial link when used in embedded controllers.

Segments (max. size each):

Code+Lists+Data+Heads+Stack+Blocks(64K)

Register use:

```
AX = -        SI = IP        CS = all segments
BX = -        DI = -         DS = CS
CX = -        BP = RP        ES = CS
DX = -        SP = SP        SS = CS
```

```
Next            Nest              Unnest
                NOP   CALL NEST
LODSW           XCHG SP,BP        XCHG SP,BP
JMP AX          PUSH SI           POP SI
                XCHG SP,BP        XCHG SI,BP
                POP SI            NEXT
                NEXT
```

```
@                   !                   Literal
POP BX              POP BX              LODSW
PUSH [BX]           POP [BX]            PUSH AX
NEXT                NEXT                NEXT
```

```
Constant            Variable            +
not implemented     NOP  CALL NEST      : +  UM+ DROP ;
                         doVAR
                    : doVar  R> ;
```

Notes: All variables are user variables, UP is in memory, FOR NEXT loop instead of DO LOOP, CATCH and THROW are used in error recovery.

F83

V2.4.0 3/24/87, Laxen & Perry

public

Written by Henry Laxen and Mike Perry to provide a working model of an 83 Standard Forth. Released with many enhancements over fig-FORTH and available for 8080/Z80, 8086 family, and 68000 series microprocessors. Includes full source code and metacompiler in DOS screen files.

An indirect-threaded 16-bit implementation with the top-of-stack on the stack. A single-segment model which interprets from screen files.

Notes: Central NEXT.

Segments (max. size each):
Code+Lists+Data+Heads+Stack+Blocks(64K)

Register use:

```
AX = -        SI = IP       CS = all
BX = W        DI = -        DS = CS
CX = -        BP = RP       ES = CS
DX = -        SP = SP       SS = CS
```

```
Next            Nest             Unnest
LODSW           INC BX           MOV SI,[BP]
MOV BX,AX       INC BX           INC BP
JMP [BX]        DEC BP           INC BP
                DEC BP           JMP NEXT
                MOV [BP],SI
                MOV SI,BX
                JMP NEXT

@               !                Literal
POP BX          POP BX           LODSW
PUSH [BX]       POP [BX]         JMP APUSH
JMP NEXT        JMP NEXT

Constant        Variable         +
INC BX          INC BX           POP BX
INC BX          INC BX           POP AX
MOV AX,[BX]     PUSH BX          ADD AX,BX
JMP APUSH       JMP NEXT         JMP APUSH
```

F-PC

V3.50 10/22/89, Zimmer & Smith

public

A massive effort (and implementation) by Tom Zimmer and Robert L. Smith (with support from a variety of other persons and groups). Many enhancements over F83 and a large set of contributed add-ons by other programmers. Has a very complete text-editor and hyper-text-like source-code and documentation browser. Very big and very complete, includes full source code and metacompiler.

A direct (segment) threaded 16-bit implementation with the top-of-stack on the stack. A multiple-segment model which interprets from text files.

Notes: colon definitions start on paragraph boundaries.

Segments (max. size each):
Code+Data+Stack+Blocks(64K)   Heads(64K)   Lists(1 meg.)

Register use:

```
AX = W        SI = IP       CS = code+data+blocks
BX = -        DI = -        DS = CS
CX = -        BP = RP       ES = Lists
DX = -        SP = SP       SS = CS
```

```
Next                  Nest                 Unnest
                      JMP NEST

LODSW ES:      NEST:  XCHG SP,BP           XCHG SI,BP
JMP AX                PUSH ES              POP SI
                      PUSH SI              POP ES
                      XCHG SP,BP           XCHG SI,BP
                      MOV DI,AX            NEXT
                      MOV AX,[DI+3]
                      ADD AX,#seg
                      MOV ES,AX
                      SUB SI,SI
                      NEXT

@                     !                    Literal
POP BX                POP BX               LODSW ES:
PUSH [BX]             POP [BX]             JMP APUSH
NEXT                  NEXT

Constant              Variable             +
JMP doCON             CALL doVAR

MOV BX,AX      doVAR: POP BX               POP BX
PUSH [BX+3]           MOV AX,[BX]          POP AX
NEXT                  PUSH BX              ADD AX,BX
                      NEXT                 JMP APUSH
```

Segments (max. size each):
Code(n*64K) Lists+Data(n*64K) Heads(n*64K) Stack(64K)

Register use:

```
AX = -        SI = IP      CS = code
BX = tos      DI = W       DS = lists+data
CX = -        BP = RP      ES = heads/misc
DX = -        SP = SP      SS = stacks
```

```
Next            Nest              Unnest
LODSW           INC BP            MOV SI,[BP]
XCHG DI,AX      INC BP            DEC BP
JMP [DI]        MOV [BP],SI       DEC BP
                LEA SI,[DI+2]     NEXT
                NEXT

@               !                 Literal
MOV BX,[BX]     POP AX            PUSH BX
NEXT            MOV [BX],AX       MOV BX,[SI]
                POP BX            INC SI
                NEXT              INC SI
                                  NEXT

Constant        Variable          +
PUSH BX         PUSH BX           POP AX
MOV BX,[DI+2]   LEA BX,[DI+2]     ADD BX,AX
NEXT            NEXT              NEXT
```

HS/FORTH
V4.24 8/9/91, Harvard Softworks commercial[1]

A very complete commercial implementation of Forth for the 8086 family of microcomputers. One of the few Forths in this review that provides compatibility with the DOS linker. Source code and metacompilers available. Multiple Forth segments in a single DOS allocation.

An indirect-threaded 16-bit implementation with the top-of-stack in a register. A multiple-segment model which interprets from text or screen files.

1. The above information is presented with the generous permission of Jim Callahan of Harvard Softworks.

Segments (max. size each):

Code(64K)	Lists(64K)	Data(64K)	Heads(64K)
Stacks(64K)	Tool(64K)	Video(64K)	Msgs(1K)

Register use:

```
AX = W        SI = IP      CS = code
BX = tos      DI = -       DS = lists
CX = -        BP = RP      ES = data
DX = -        SP = SP      SS = stacks
```

```
Next            Nest              Unnest
LODSW           DEC BP            MOV SI,[BP]
JMP AX          DEC BP            INC BP
                MOV [BP],SI       INC BP
                MOV SI,pfa        NEXT
                NEXT

@               !                 Literal
MOV BX,ES:[BX]  POP ES:[BX]       PUSH BX
NEXT            POP BX            MOV BX,value
                NEXT              NEXT

Constant        Variable          +
PUSH BX         PUSH BX           POP AX
MOV BX,value    MOV BX,addr       ADD BX,AX
NEXT            NEXT              NEXT
```

KForth
V0.9 9/28/91, Kelly copyrighted

Currently an experimental model to investigate various aspects of threading and segmentation. Current version is fast ("in-line") direct-threaded, multi-segment (in multiple DOS segments).

A direct-threaded 16-bit implementation with the top-of-stack in a register. A multiple-segment model which interprets from screen files.

Notes: Colon-word PFA's, literals, constants, and variable addresses are compiled "in-line" in the code segment.

LaFORTH
V4.0 9/24/87, Stuart & Smith
copyrighted

Experimental version by LaFarr Stuart and Robert L. Smith. Has some very interesting features (including calling Forth from Forth and interpreting a word-at-a-time instead of a line-at-a-time). Source code is in assembler.

A direct-threaded 16-bit implementation with the top-of-stack on the stack. A single-segment model with an extra segment for interpreting text files.

Notes: the return stack is in the ES segment and grows "up."

Segments (max. size each):
Code+Lists+Data+Heads+Stack(64K)   Text(64K)

Register use:

```
AX = W        SI = IP        CS = code+lists+data+p-stack
BX = -        DI = RP        DS = CS
CX = -        BP = -         ES = CS+1000H txt-buf+r-stack
DX = -        SP = SP        SS = CS
```

```
Next              Nest                Unnest
                  JMP NEST
LODSW       NEST: ADD AX,3            SUB DI,2
JMP AX            XCHG AX,SI          MOV SI,ES:[DI]
                  STOSW               NEXT
                  NEXT

@                 !                   Literal
POP BX            POP BX              LODSW
PUSH [BX]         POP AX              PUSH AX
NEXT              MOV [BX],AX         NEXT
                  NEXT

Constant          Variable            +
CALL @            CALL @              POP AX
                                      POP BX
                                      ADD AX,BX
                                      PUSH AX
                                      NEXT
```

MMSFORTH
V2.4 5/30/85,
Miller Microcomputer Svcs.
commercial[1]

Commercial version of Forth includes advanced full-screen editor, many utilities. Options include database, word-processor, general ledger, expert system, and advanced utilities. Source-code is in screen files in DOS version and in direct blocks (screens) in self-booting version (which supports more efficient Forth disk formats such as 1K sector size). Most source-code is supplied, full source-code and metacompiler are available.

Indirect-threaded, 16-bit implementation, top-of-stack on stack, single-segment model (DOS version uses a separate Heads segment). Interprets from direct blocks (DOS version uses screen-files.)

Segments (max. size each):
Code+Lists+Data+Heads+Stack+Blocks(64K) (non-DOS version)

Register use:

```
AX = -        SI = IP        CS = all
BX = W        DI = -         DS = CS
CX = -        BP = RP        ES = CS
DX = -        SS = SP        SS = CS
```

```
Next              Nest                Unnest
LODSW             DEC BP              MOV SI,[BP]
XCHG AX,BX        DEC BP              INC BP
JMP [BX]          MOV SI,[BP]         INC BP
                  INC BX              NEXT
                  INC BX
                  MOV SI,BX
                  NEXT

@                 !                   Literal
POP BX            POP BX              *
PUSH [BX]         POP [BX]
NEXT              NEXT

Constant          Variable            +
*                 *                   POP AX
                                      POP DX
                                      ADD AX,DX
                                      PUSH AX
                                      NEXT
```

Note: Non-DOS version of MMSFORTH was used in Excalibur's SAVVY, DOS version in Lindberg System's OMNITERM-2 and Ashton-Tate's RAPIDFILE.

* High-level words, source code provided.

1. The above information is presented with the generous permission of A. Richard (Dick) Miller of Miller Microcomputer Services.

MVP-FORTH

V1.0405.03 5/17/85, MVP
public

One of the first 79-Standard Forths. Based on the fig-FORTH 8086 implementation model. Source code in direct blocks (screens).

An indirect-threaded, 16-bit implementation with the top-of-stack on the stack. A single-segment model which interprets from direct blocks (screens).

Notes: NEXT and NEST seem to be a direct translation of the fig-FORTH 8080 assembly code. Central NEXT.

Segments (max. size each):
Code+Lists+Data+Heads+Stack+Blocks(64K)

Register use:

```
AX = -        SI = IP      CS = all
BX = -        DI = -       DS = CS
CX = -        BP = RP      ES = -
DX = W        SP = SP      SS = CS
```

```
Next           Nest           Unnest
MOV AX,[SI]    INC DX         MOV SI,[BP]
INC SI         DEC BP         INC BP
INC SI         DEC BP         INC BP
MOV BX,AX      MOV [BP],SI    JMP NEXT
MOV DX,AX      MOV SI,DX
INC DX         JMP NEXT
JMP [BX]

@              !              Literal
POP BX         POP BX         MOV AX,[SI]
MOV AX,[BX]    POP AX         INC SI
JMP APUSH      MOV [BX],AX    INC SI
               JMP NEXT       JMP APUSH

Constant       Variable       +
                              POP AX
                              POP BX
                              ADD AX,BX
                              JMP APUSH
```

PC-Forth

V1.56 9/09/87, Kelly
public

Written expressly for teaching purposes. Includes full source code and a simple, interactive metacompiler (which does not require mixing meta-commands in with the Forth source code). Also includes a very powerful screen editor (with overlay capability for use with other Forths), as well as many utilities. Available in both self-booting and DOS versions.

An indirect-threaded 16-bit implementation with the top-of-stack on the stack. A single-segment model which interprets from screens (direct or file) or text files.

Segments (max. size each):
Code+Lists+Data+Heads+Stack+Blocks(64K)

Register use:

```
AX = -        SI = IP      CS = all
BX = W        DI = -       DS = CS
CX = -        BP = RP      ES = CS
DX = -        SP = SP      SS = CS
```

```
Next           Nest           Unnest
LODSW          DEC BP         MOV SI,[BP]
XCHG AX,BX     DEC BP         INC BP
JMP [BX]       MOV [BP],SI    INC BP
               INC SI         NEXT
               INC SI
               MOV SI,BX
               NEXT

@              !              Literal
POP BX         POP BX         LODSW
PUSH [BX]      POP [BX]       PUSH AX
NEXT           NEXT           NEXT

Constant       Variable       +
INC BX         INC BX         POP AX
INC BX         INC BX         POP DX
PUSH [BX]      PUSH BX        ADD DX,AX
NEXT           NEXT           PUSH DX
                              NEXT
```

polyFORTH
pF86S/MSD, FORTH Inc. commercial[1]

The mother of all Forths (well, almost) by FORTH, Inc. Complete source code with metacompiler, EGA/VGA graphics, data-base, floating point, screen editor, debugger and other support. Full multi-user capability built in at the kernel level. Source code and shadow screens in screen files.

Indirect-threaded, 16-bit implementation, top-of-stack on stack, multiple-segment model. Interprets from screen files. A 32-bit 386 protected-mode version is also available.

Segments (max. size each):
Code+Lists+Heads(n*64K) Data+Stack+Blocks(64K) Extended-data(M)

Register use:

```
AX = -        SI = IP      CS = code+lists+heads
BX = U        DI = W       DS = data+stacks
CX = -        BP = RP      ES = -
DX = -        SP = SP      SS = DS
```

Next
```
CS: LODSW
XCHG AX,DI
JMP [DI]
```

Nest
```
XCHG SP BP
PUSH SI
XCHG SP BP
LEA CELL SI,[DI]
NEXT
```

Unnest
```
XCHG SP BP
POP SI
XCHG SP BP
NEXT
```

@
```
POP DI
PUSH [DI]
NEXT
```

!
```
POP DI
POP [DI]
NEXT
```

Literal
```
CS: LODSW
PUSH AX
NEXT
```

Constant
```
MOV CS: DI,[DI+2]
PUSH [DI]
NEXT
```

Variable
```
MOV CS: DI,[DI+2]
PUSH [DI]
NEXT
```

+
```
POP DX
POP AX
ADD AX,DX
PUSH AX
NEXT
```

Note: the reported benchmarks were done on pF86/MSD (which is a single-segment version dated 1/20/87). The newer, multi-segment version detailed above should produce the same or only slightly different times.

1. The above information is presented with the generous permission of Elizabeth Rather of FORTH, Inc.

Pygmy
V1.3 10/4/90, Sergeant copyrighted

Based on the Chuck Moore cmFORTH model. The source code and the metacompiler are in screen file.

A direct-threaded 16-bit implementation with the top-of-stack in a register. A single-segment model which interprets from screen files.

Segments (max. size each):
Code+Lists+Data+Heads+Stack+Blocks(64K)

Register use:

```
AX = W        SI = IP      CS = all
BX = tos      DI = -       DS = CS
CX = -        BP = RP      ES = CS
DX = -        SS = SP      SS = CS
```

Next
```
LODSW
JMP AX
```

Nest
```
        JMP NEST

NEST:   XCHG SP,BP
        PUSH SI
        XCHG SP,BP
        ADD AX,3
        MOV SI,AX
        NEXT
```

Unnest
```
XCHG SP,BP
POP SI
XCHG SI,BP
NEXT
```

@
```
MOV BX,[BX]
NEXT
```

!
```
POP AX
MOV [BX],AX
POP BX
NEXT
```

Literal
```
PUSH BX
LODSW
MOV BX,AX
NEXT
```

Constant (in-line)
```
PUSH BX
MOV BX,value
NEXT
```

Variable
```
        JMP doVAR

doVAR:  PUSH BX
        ADD AX,3
        MOV BX,AX
        NEXT
```

+
```
POP AX
ADD BX,AX
NEXT
```

riFORTH
V1? 1990, Illyes
copyrighted

A minimalist Forth; fast and efficient. Full source code and metacompiler in about 15 screens. Does some optimization. Interesting!

A subroutine-threaded, 16-bit implementation with the top-of-stack in a register. A single-segment model which interprets from screen files.

riFORTH Copyright Robert F. Illyes, 1990. My thanks to Robert Illyes for publishing the source code for riFORTH. The availability of a complete subroutine-threaded Forth, in only 12 screens, made it possible to done the five different versions used in this study.

<u>Segments (max. size each):</u>
Code+Lists+Data+Heads+Stack+Blocks(64K)

<u>Register use:</u>

AX = -	SI = SP	CS = all	IP = IP
BX = tos	DI = -	DS = CS	
CX = -	BP = -	ES = CS	
DX = -	SS = RP	SS = CS	

Next	Nest	Unnest
	CALL xxx	RET

@	!	Literal
(XCHG SP,SI)*	(XCHG SP,SI)*	(XCHG SP,SI)*
MOV BX,[BX]	POP [BX]	PUSH BX
(XCHG SP,SI)*	POP BX	MOVE BX,value
	(XCHG SP,SI)*	(XCHG SP,SI)*

Constant	Variable	+
(XCHG SP,SI)*	(XCHG SP,SI)*	(XCHG SP,SI)*
PUSH BX	PUSH BX	POP AX
MOV BX,value	MOV BX,addr	ADD BX,AX
(XCHG SP,SI)*	(XCHG SP,SI)*	(XCHG SP,SI)*

* When these words are compiled in-line, these instructions may be eliminated.

Illyes, Robert F., "A Tiny and Very Fast Subroutine-threaded Forth", *Proceedings of the 1990 Rochester Forth Conference*, page 76, The Forth Institute.

Upper Deck
V2.0 1/26/91,
Upper Deck Systems
commercial[1]

An inexpensive, powerful commercial version of Forth which uses multiple DOS segments. It includes a very nice resident text-editor.

A direct-threaded 16-bit implementation with the top-of-stack in a register. A multiple-segment model which interprets from text files.

Notes: When case sensitive, all supplied words are lower case.

1. The above information is presented with the generous permission of Peter Graves of Upper Deck Systems.

<u>Segments (max. size each):</u>

Code(64K)	Lists(64K)	Data(64K)	Heads(64K)	Stacks(64K)

<u>Register use:</u>

AX = W	SI = IP	CS = code
BX = tos	DI = W'	DS = lists
CX = -	BP = RP	ES = data
DX = -	SS = SP	SS = stacks

Next		Nest	Unnest
		MOV DI,pfa	
		JMP NEST	
LODSW	NEST:	DEC BP	MOV SI,[BP]
JMP AX		DEC BP	INC BP
		MOV [BP],SI	INC BP
		MOV SI,DI	NEXT
		NEXT	

@	!	Literal
MOV ES:BX,[BX]	POP ES:[BX]	PUSH BX
NEXT	POP BX	MOV BX,[SI]
	NEXT	INC SI
		INC SI
		NEXT

Constant		Variable	+
		MOV DI,addr	
		JMP doVAR	
PUSH BX	doVAR:	PUSH BX	POP AX
MOVE BX,value		MOV BX,DI	ADD BX,AX
NEXT		NEXT	NEXT

UR/FORTH
V1.1 3/11/90, LMI commercial[1]

UR/FORTH is one of the few Forths in this review that is compatible with the DOS linker. It is well supported, with many extensions and a very good screen-oriented editor. Most source code is provided in screen files. Complete source is available.

A direct-threaded 16-bit implementation with the top-of-stack in a register. A multiple DOS-segment model which interprets from screen or text files. Also available in OS/2 1.x, 386 32-bit protected mode, and Windows implementations (which are compatible, at Forth language level, with the DOS version).

Note: supports binary overlays.

Segments (max. size each):
Code(64K) Lists+Data(64K) Heads(64K) Stacks(64K)[2]

Register use:

```
AX = W          SI = IP         CS = code
BX = tos        DI = -          DS = lists+data
CX = -          BP = RP         ES = -
DX = -          SS = SP         SS = stacks
```

```
Next                Nest                 Unnest
                    MOV DI,pfa
                    JMP NEST
LODSW         NEST: XCHG SP,BP           MOV SI,[BP]
JMP AX              PUSH SI              INC BP
                    XCHG SP,BP           INC BP
                    MOV SI,DI
                    NEXT                 NEXT

@                   !                    Literal
MOV BX,[BX]         POP [BX]             PUSH BX
NEXT                POP BX               LODSW
                    NEXT                 MOV BX,AX
                                         NEXT

Constant            Variable             +
MOV DI,value        MOV DI,addr
JMP doCON           JMP doVAR
PUSH BX      doVAR: PUSH BX              POP AX
MOV BX,DI           MOV BX,DI            ADD BX,AX
NEXT                NEXT                 NEXT
```

1. The above information is presented with the generous permission of Ray Duncan of Laboratory Microsystems, Inc.
2. Segment model for version tested, varies with implementation.

ZEN
V1.5a 4/2/91, Tracy copyright

Currently (Sept. 1991) ZEN is the only Forth in this review that is tracking the X3J14 basis document. It has fully ROMmable assembler source code and an interface to one of the standard programmers text editors.

A direct-threaded 16-bit implementation with the top-of-stack in a register. A multiple-segment model which interprets from screen or text files.

Segments (max. size each):
Code+Lists(64K) Data+Heads+Stack(64K)

Register use:

```
AX = W          SI = IP         CS = code+lists
BX = tos        DI = -          DS = data+heads+stacks
CX = -          BP = RP         ES = -
DX = -          SS = SP         SS = DS
```

```
Next                Nest                 Unnest
                    CALL NEST
LODSW CS:     NEST: DEC BP               MOV [SI],BP
JMP AX              DEC BP               INC BP
                    MOV [BP],SI          INC BP
                    POP SI               NEXT
                    NEXT

@                   !                    Literal
MOV BX,[BX]         POP [BX]             LODSW
NEXT                POP BX               PUSH BX
                    NEXT                 MOV BX,AX
                                         NEXT

Constant            Variable             +
JMP doCON           JMP doVAR            POP AX
                                         ADD BX,AX
See variable doVAR: PUSH BX              NEXT
                    ADD AX,3
                    XCHG AX,BX
                    MOV BX,CS:[BX]
                    NEXT
```

PART ONE

The Curly Control Structure Set

Kourtis Giorgio

Genoa, Italy

AUTHOR
(smiling)
Hi! I've got a new proposal on a new complete set of control structures.

READER
(annoyed)
Another proposal? Do you know that this is the 134th Forth proposal on extensions, expansions, additions, etc., to control structures?

AUTHOR
(less smilingly)
Yes, but mine...

READER
(smiling)
Oh, yes! Sure, yours is better, includes as subcases all previous proposals, is original, has support for errors, is coherent, etc.

AUTHOR
(happy)
Exactly!

READER
(serious)
You know that the same thing has been claimed by 59 other articles?

AUTHOR
(aggressive)
Yes, I know, I have read every article. Many are very interesting and have been important for me. I copied everything that could be copied, I took every good idea that has appeared, I tried to unify solutions, I attempted to solve all the problems I was aware of,* I tried to render uniform the proposed set of words, I sacrificed strict historical continuity to improve teachability while avoiding conflicts among old and new syntaxes. And I have used them for more than two years, refining them until they were stable enough.

* except the interactivity of control-flow words.

READER
(very dubious)
Hmm... okay, I'll listen to you, but I hope you'll have something new to tell me.

AUTHOR
(happy and enthusiastic)
Oh, thank you! I'll present six control structures that you can add at will. Here are some simple rules to follow:

Every control structure has a name, a beginning, and an end. The beginning is set by the word `name{` while the end is set by the word `name}`. E.g.,

```
CASE{   CASE}
LOOP{   LOOP}
```

The beginning of a control structure may accept some value on the stack, e.g.,

```
5 TIMES{ ... TIMES}
```

The end of the control structure, when reached, may jump out of the control structure, or can jump unconditionally or conditionally to the beginning of the control structure (more precisely, to the word immediately after the beginning).

`CONTROL}` and `CASE}` jump out, `REPEAT}` and `FOR}` jump unconditionally to the beginning, while `TIMES}` and `LOOP}` jump conditionally to the beginning or out of the control structure.

With the exception of `CONTROL` and `REPEAT`, the other control structures dispose of an index (like a `DO LOOP` in standard Forth). That index (usually named `I`) has a different meaning among control structures. (In the `CASE` control structure, for example, `I` contains the subject of our research.)

As shown in Figure One-a, when inside a control structure, we can use some control-flow words that jump conditionally or unconditionally beyond the end of the control structure (like `LEAVE`, `WHEN`, and `WHILE`) or can jump to the word immediately after the beginning of the control structure (like `AGAIN` and `?AGAIN`).

These control-flow words can be used any number of times in any combination, and also can be used in secondaries called from within the control structure—allowing, for example, WHILE to be defined as

```
: WHILE ( flag -- )
  0= WHEN  ;
```

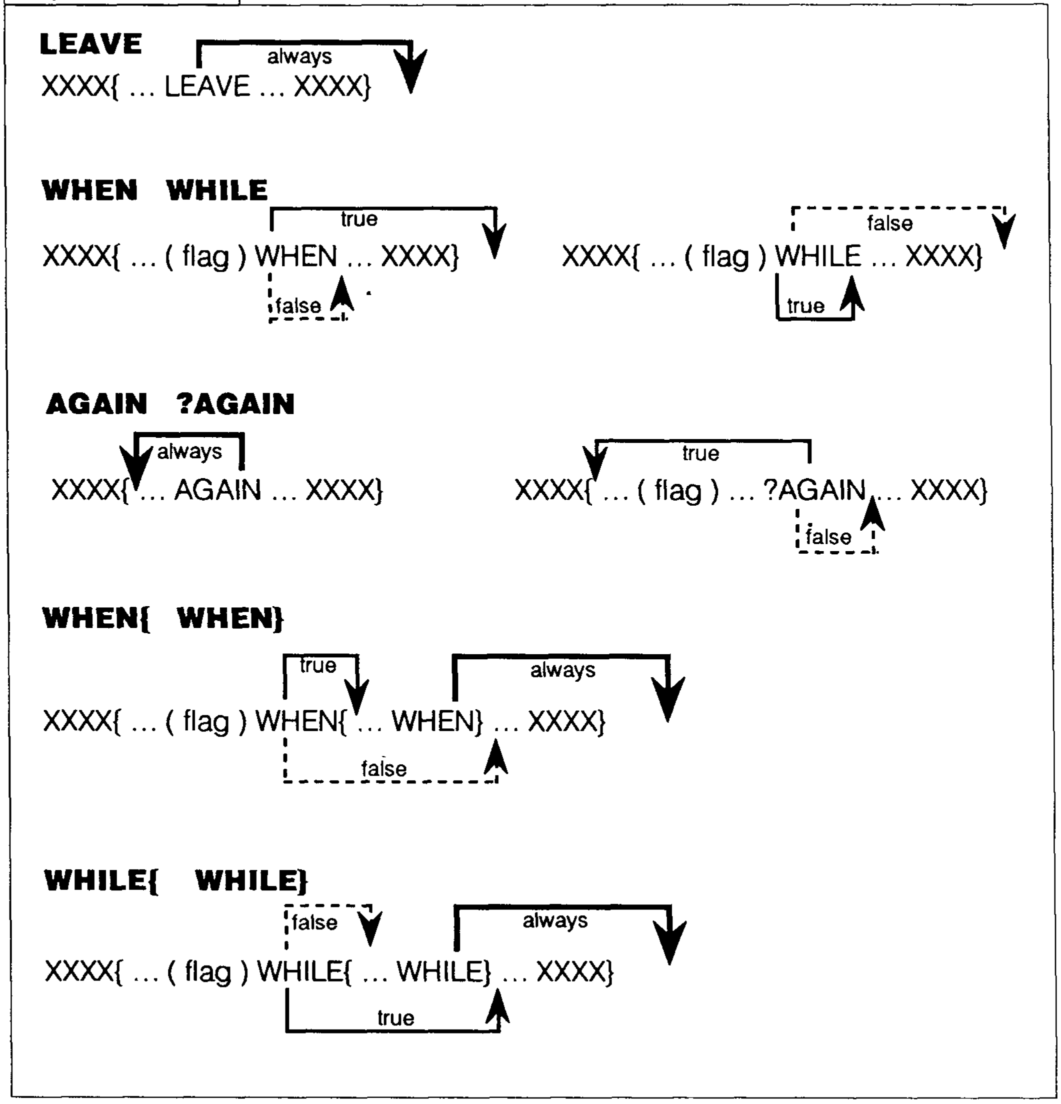
Depending on the implementation, this feature may be available or not. While not often used, it is a rarely available but interesting feature that can provide new possibilities.

Looking again at Figure One-a, we see that apart from LEAVE, WHEN, WHILE, AGAIN, and ?AGAIN, there are words that must be used in pairs:

```
WHEN{   ( flag -- )
WHEN}  ( -- )

WHILE{   ( flag -- )
WHILE}  ( -- )
```

Figure One-a.



In brief, if xxxx is any control structure,
xxxx{ ... WHEN{ someCode WHEN} ... xxxx}

is equivalent to

```
xxxx{ IF someCode LEAVE THEN ... xxxx}
```

and, similarly,

```
xxxx{ WHILE{ someCode WHILE} ... xxxx}
```

is equivalent to

```
xxxx{ ... 0= IF someCode LEAVE THEN ... xxxx}
```

By using a WHEN pair, we not only can test a condition to decide if the control structure must be left, we can also place between WHEN{ and WHEN} the code that must be executed just before leaving the control structure. This way, any leave-test point can be equipped with its own, specific pre-exit code.

Using WHEN is, thus, the equivalent of writing WHEN{ and WHEN} with no code between them.

To illustrate figuratively, let's say that:

```
xxxx{ WHEN{ someThing WHEN} xxxx}
```

may be informally written as

```
xxxx{    WHEN    xxxx}
           someThing
```

Other pairs of words that we can use inside an indexed control structure are:

```
OF{     ( value -- )
OF}     ( -- )

WITHIN{      ( lower upper -- )
WITHIN}      ( -- )

IN{     ( x1 x2 ... xN N -- )
IN}     ( -- )
```

These pairs make explicit reference to the index and are mainly used in the internals of a CASE structure (though it may be used in any indexed control structure). Look at Figure One-b for a definition of their workings.

Inside indexed control structures, three words are supplied to reference the index and its value:

```
I       ( -- indexValue )
```

Leaves on the stack the value of the index.

`TO-I ( newValue -- )`
Stores into the index a new value.

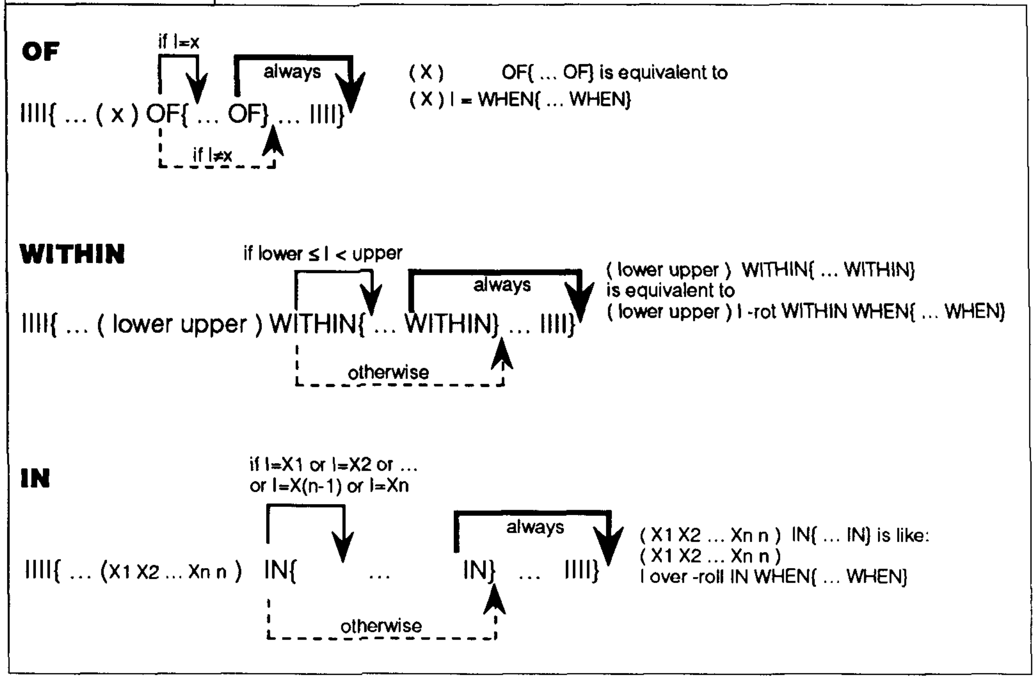
`STEP ( valueToAdd -- )`
Adds to the index a value (stepping it). `STEP` is equivalent to `I + TO-I`.

Description of Curly Control Structures

Given the previous framework, now I am going to illustrate briefly the use of each of these control structures and how they work. Later, we will see some examples.

```
CONTROL{   ( -- )
CONTROL}   ( -- )
```

Figure One-b.



The word `CONTROL{` marks the beginning of the control structure. The word `CONTROL}` marks the end. They may be used as label points when jumping via words like `WHEN`, `WHILE`, `AGAIN`, `?AGAIN`, and word pairs like `WHEN{ ... WHEN}`, etc.

`ANDIF` structures are easily implementable with these ingredients. The word `CONTROL}`, if reached, exits the control structure.

```
REPEAT{      ( -- )
REPEAT}      ( -- )
```

Equivalent to—but much more flexible than—the usual `BEGIN UNTIL` or `BEGIN WHILE REPEAT` structures. Any number of `WHILE`s or `WHEN`s may be used inside this structure, along with the pairs `WHEN{ WHEN}`, etc.

```
CASE{  ( KeyValue -- )
CASE}  ( -- )
```

Comparable to Eaker's `CASE` structure. `CASE{` takes a number from the stack (the subject of our research) and puts it into the index, where it can be retrieved by

`I      ( -- KeyValue )`

Afterwards, by means of pairs like `OF{ ... OF}`, `WITHIN{ ... WITHIN}`, `IN{ ... IN}`, and others yet to be invented, and also by using `WHEN{ ... WHEN}` or `WHILE{ ... WHILE}`, we can select and perform the desired action. If the `CASE}` word is reached, the control structure is left.

```
FOR{   ( initialValue -- )
FOR}   ( -- )
```

General-purpose looping construct (unlike a `FOR ... NEXT` definite loop). `FOR{` takes a number from the stack and puts it into the index. Afterwards, the index may be manipulated by words like `I`, `TO-I`, and `STEP`. The loop-termination condition must be handled explicitly using `WHILE` or `WHEN`.

This is an imitation of the C language's `FOR` construct. The desired model of loop (pre-increment, pre-decrement, post-increment, post-decrement, fixed or variable step, etc.), must be handled explicitly—checking and incrementing/decrementing at the beginning or end of the loop, depending on the desired behavior. If `FOR}` is reached, the loop is repeated.

```
TIMES{ ( #times -- )
TIMES} ( -- )
```

Similar to the `0 DO ... LOOP` construct. Takes a number from the stack (there exists an error condition if the number is negative) and puts it into the index.

a) Before every iteration, the value of the index is checked. If `I` contains 0, the control structure is left and execution continues after `TIMES}`. Otherwise, `I` is decremented and execution continues after `TIMES{`, beginning an iteration. If and when `TIMES}` is reached, the process is repeated from point a) (pre-incrementing model).

```
LOOP{  ( Start #times step -- )
LOOP}  ( -- )
```

Start specifies the initial value of the index, #times specifies the maximum number of times the loop must be done (it could end prematurely due to words like `LEAVE`, `WHEN`, and `WHILE`).

If #times is negative, there is an error condition. Step and #times are put on the return stack, and start is put into the index (also on the return stack).

a) Before every iteration, the #times is checked. If it is 0, the control structure is left and execution proceeds after `LOOP}`. If #times is not 0, it is decremented and an iteration begins.
If and when `LOOP}` is reached, the step is added to the index, then the process is repeated from point a).

Thus,

```
10 ( beginning )
4  ( times )
```

```
5  ( step )
LOOP{ I . LOOP}
```

types 10 15 20 25, while

```
20 ( beginning )
4  ( times )
-3 ( step )
LOOP{ I . LOOP}
```

types 20 17 14 11

Along with the loop structure are furnished the four parameter-modifying words END, END], SIZE, and BACK. These words allow you to specify, in many different ways, the order and the set of values that must be spanned by the index during the loop.

Examples and Test Suites

The above set of control-flow words has been presented by figures and somewhat by words. Examples are important for two reasons:

- Clarify obscure or dubious points.
- Given the fact that only the constructing elements have been shown, provide some interesting combinations of them. (Some, probably, haven't even been explored yet.)

CONTROL

Example: Test whether the three variables A, B, and C contain 0. (The so-called ANDIF construct.)

```
: ABC_allZero?
  CONTROL{  A @ 0= WHILE
            B @ 0= WHILE
            C @ 0= WHILE
  ." A,B,C contain 0"  CONTROL}  ;
```

REPEAT

Example One: Traverse a list until a 0 list terminator is found.

```
( addr -- ) REPEAT{
DUP @ WHILE REPEAT} ( lastAddr )
```

or, equivalently:

```
( addr -- ) REPEAT{
DUP @ 0= WHEN @ REPEAT}
```

Example Two: Given the address of a null-terminated string, leave the address of the first space or the EndAddr (if no spaces are found).

```
( addr -- ) REPEAT{
DUP C@ ?DUP
WHILE BL = WHEN 1+ REPEAT}
```

CASE

Example One: Take a character code from the stack and qualify it.

```
: ?WhatCharItIs?   ( char -- )
 CASE{ ( pop char from stack and put into index)
   ." The character with code" I . ." is "
ASCII A  ASCII Z  []
WITHIN{ ." an upper-case letter" WITHIN}
ASCII a  ASCII z  []
WITHIN{ ." a lower-case letter" WITHIN}
  \ The word [] means simply 1+

ASCII +  ASCII -  ASCII *
ASCII /  ASCII ^ 5
 \ 5 specifies that +, -, *, /, and ^ are 5.
IN{  ." an arithmetic operator" IN}

ASCII (  ASCII [ ASCII {  3
IN{ ." an opening parenthesis"  IN}
ASCII )  ASCII ]  ASCII }  3
IN{ ." a closing parenthesis"  IN}

BL OF{  ." SPACE char" OF}
 0 OF{  ." NULL char"  OF}

ASCII 0  ASCII 9 []
WITHIN{ ." a decimal digit" WITHIN}
[ HEX ]  80 100
WITHIN{
." a graphical character. One of those " cr
." with code between 128 included and 256 excluded"
WITHIN}

I 100 >=  I 0 <  OR
WHEN{ ." outside the character range" WHEN}

I -20 AND
WHILE{ ." a control character" WHILE}
  \ -20 = not (11111B)

." unclassified"
CASE}
( CASE} if reached leaves the control structure)
;
```

Note: The two words ((and)) may be defined to count the elements of a set of numbers. So, instead of writing

```
ASCII (  ASCII [  ASCII { 3
```

we can write

```
(( ASCII (  ASCII [  ASCII {  ))
```

and this is much better. See the provided code for their definitions.

Example Two: Special use of the case structure which allows for subcases. Pairs used inside the outermost WITHIN, if entered, leave the CASE structure and execution continues after CASE} (just like external pairs).

```
: DISASSEMBLE  ( inst -- )
  CASE{
    0000 4000 WITHIN{
    I MoveInst WITHIN}
```

```
4000 8000 WITHIN{
  5000 6000 WITHIN{
  I AddInst WITHIN}
  6000 7000 WITHIN{
  I SubInst WITHIN}
I 1 AND 0= WHEN{
I JsrInst  WHEN}
( else ) I JmpInst
WITHIN}

I 8000 - 1000 / TO-I
1 OF{ ... OF}
...
7 OF{ ... OF}

CASE} ResultDisplay ;
```

FOR

Example: Type the powers of two smaller than 1.000.000

```
1 FOR{  I 1.000.000 < WHILE
        I .
        I 2* TO-I  ( or I STEP ) FOR}
```

Example: Type the contents of a null-terminated list.

```
( StartAddr -- )
FOR{ I WHILE
     I CELL+ @ .
     I @ TO-I FOR}
```

Example: Do a loop that executes at least once (like DO LOOP). Only for illustration purposes.

```
10 FOR{  someCode
 5 STEP I 90 <
   WHILE FOR}
90 10 DO someCode
 5 +LOOP
```

TIMES

```
: DROPS
  ( X1 X2 ... Xn n --)
  TIMES{ DROP
  TIMES} ;

: MULTIEXECUTE
  ( token #times --)
  TIMES{
  DUP EXECUTE
  TIMES} DROP ;

: CountBack (from --)
  TIMES{  I .
  TIMES} ;
```

`5 CountBack`
types 4 3 2 1 0

`1 CountBack`
types 0

`0 CountBack`
doesn't type anything.

`-2 CountBack`
issues an error message.

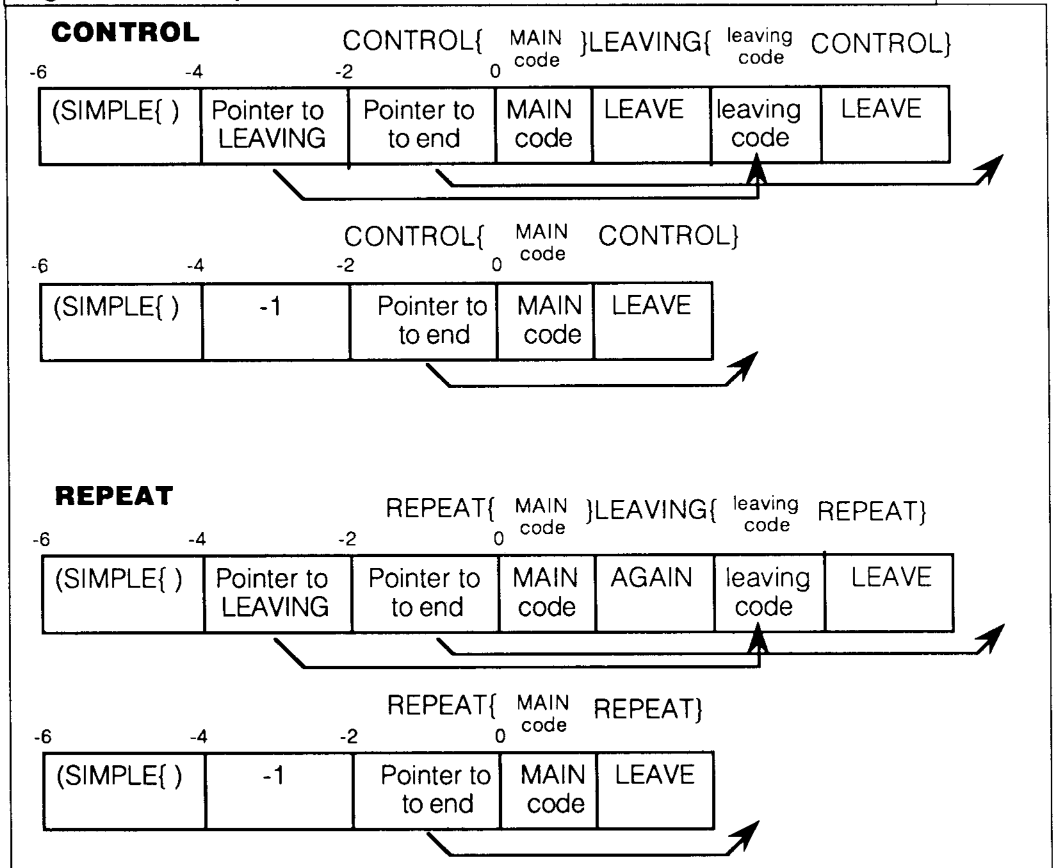
Presenting }LEAVING{ and }COMPLETED{

While the subset of words discussed above resolves many problems, until now some are still unresolved. Here I will present the problems and the solutions to them using extensions of the wordset.

The }LEAVING{ Clause

Let's suppose there are three variables V0, V1, and V2 that contain addresses of strings, and we want to know if at least two among them are equal. Using the set presented so far, here is a solution:

Figure Two. Compilation effects of CONTROL, REPEAT, FOR, and CASE.



FOR & CASE
CASE and FOR compile similarly to CONTROL and REPEAT, respectively, but with (SIMPLE) replaced by (INDEXED).

```
CONTROL{  V0 @ V1 @ $= WHEN{
               ." At least two equal" WHEN}
          V1 @ V2 @ $= WHEN{
               ." At least two equal" WHEN}
          V2 @ V0 @ $= WHEN{
               ." At least two equal" WHEN}
          ." All different "
CONTROL}
```

This solution, however, is redundant and wasteful of space. At the expense of computational time, we could choose to check all three equalities by ORing them together at the end and using an IF ELSE THEN control structure. Here, I'll show a third solution that is neither redundant nor slow.

```
CONTROL{  V0 @ V1 @ $= WHEN
          V1 @ V2 @ $= WHEN
          V2 @ V3 @ $= WHEN
            ." All different"
          }LEAVING{ ." At least two equal"
CONTROL}
```

The previous solution and its meaning may be explained in more general terms.

Let XXXX be the name of a control structure (CONTROL, TIMES, etc.). Let's call the pair of words like WHEN{ ... WHEN}, WHILE{ ... WHILE}, OF{ ... OF}, etc. as specialized leaving points, while words like WHEN and WHILE are unspecialized leaving points. Thus, the code in Figure Seven is logically equivalent to that in Figure Eight.

In other words, suppose we have a control structure with a }LEAVING{ embedded in it. If we transform all unspecialized leaving points into the corresponding specialized pair—inserting into the specialized pair the code contained between the original }LEAVING{ and the end of the control structure, then deleting the code from }LEAVING{ (inclusive) to the end of the control structure (exclusive)—we obtain new code that is logically equivalent to the original. (Again, refer to Figures Seven and Eight.)

Another example will better clarify these concepts. Suppose we have a null-terminated string and must scan it for the first occurrence of the character +, –, or .. If such an occurrence is found, we must substitute a space for it and leave on the stack (SubstitutionAddr+1 true); otherwise, we must leave a (false) on the stack. Here is a solution using }LEAVING{:

```
( StringAddr )
FOR{  I C@ WHILE{  false WHILE}
      I C@ ascii +  = WHEN
      I C@ ascii -  = WHEN
      I C@ ascii .  = WHEN  1 STEP
      }LEAVING{ bl I C!  I 1+ true
FOR}
```

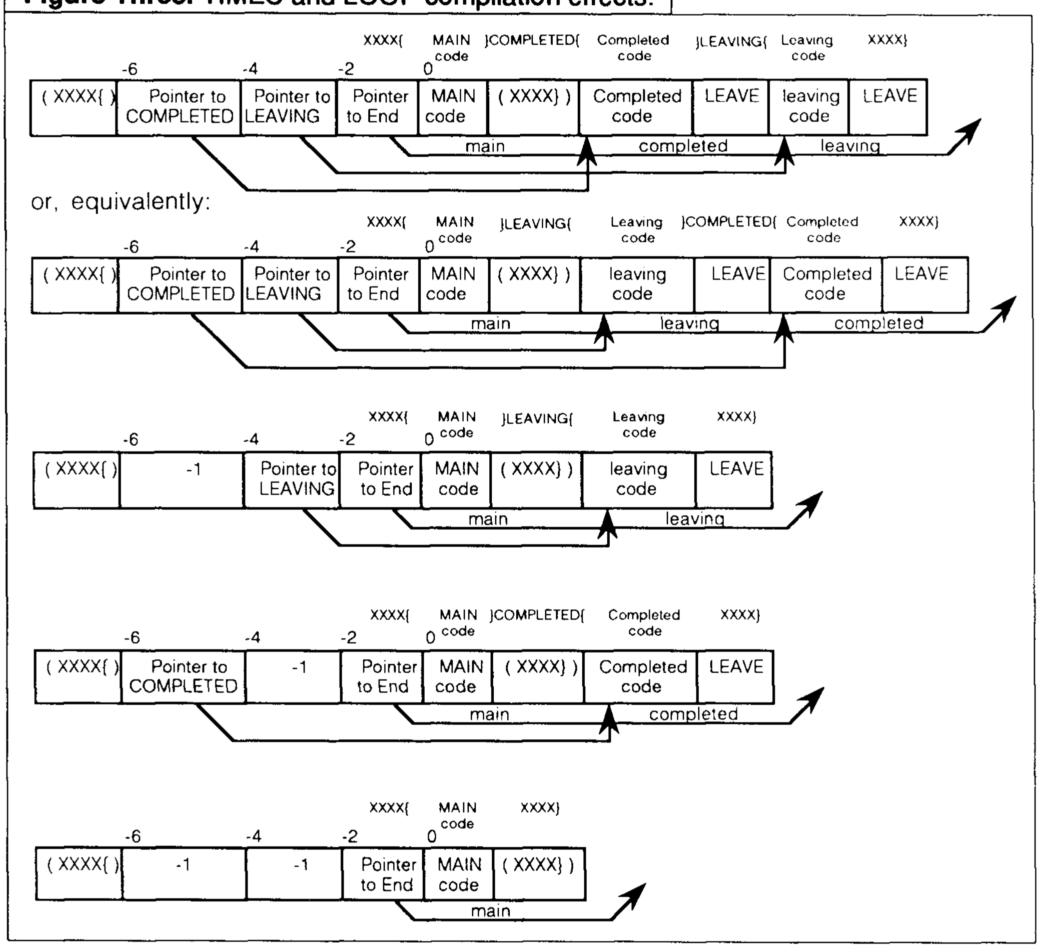
Maybe the above would read more clearly if written informally as:

```
FOR{  I C@ WHILE{  false WHILE}
      I C@ ascii +  = ORIF
      I C@ ascii -  = ORIF
      I C@ ascii .  = ORIF  1 STEP
      }ORWHEN{ bl I C!  I 1+ true
FOR}
```

The }COMPLETED{ Clause

A similar problem is encountered in definite loops. A definite loop like TIMES or LOOP may end for two reasons: 1. Premature end due to "leavers" like WHEN, WHILE, LEAVE,

Figure Three. TIMES and LOOP compilation effects.



etc.

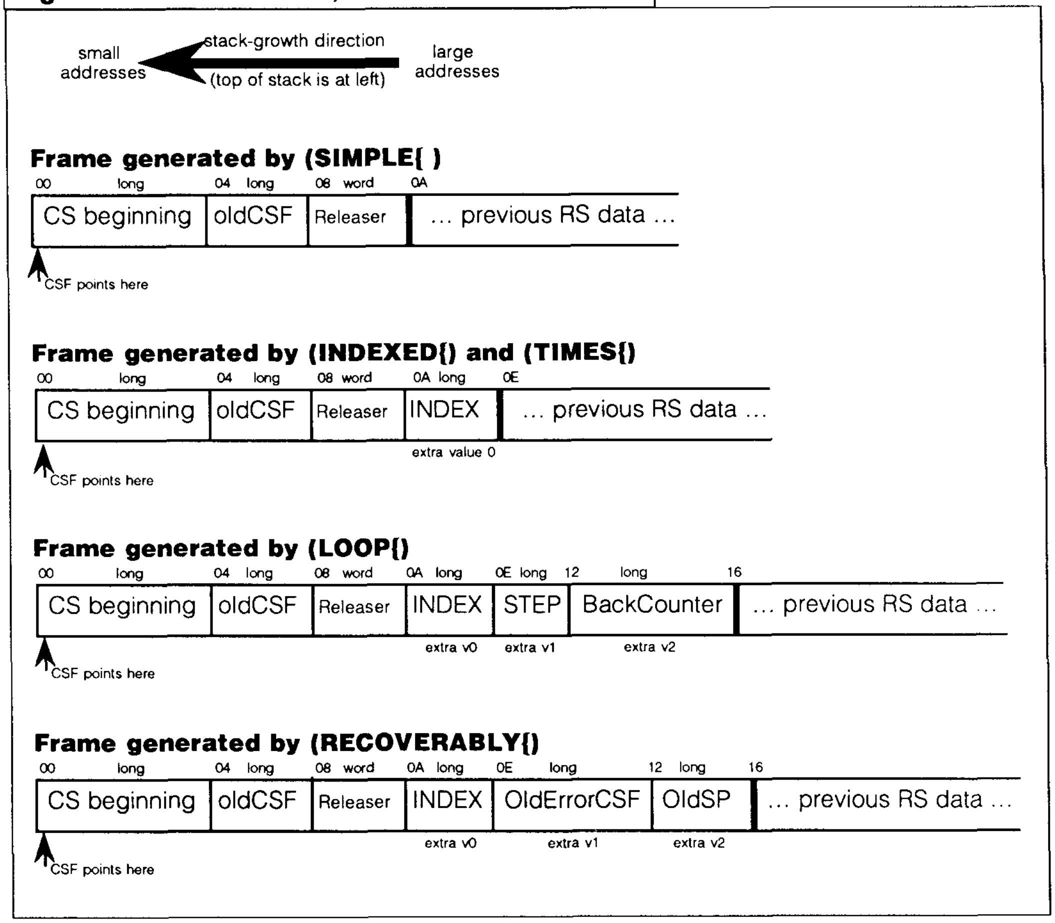
2. Exhaustion of the number of looping times specified.

Depending on the reason why execution left the loop, different behaviors can be requested. Let's explain via example.

Suppose we must search an editor's text for a specific character. We want the cursor to move during the search and, if the character is found, have it point to that character; otherwise, we want to reset the cursor to its original position, not simply leaving it at the end of the text.

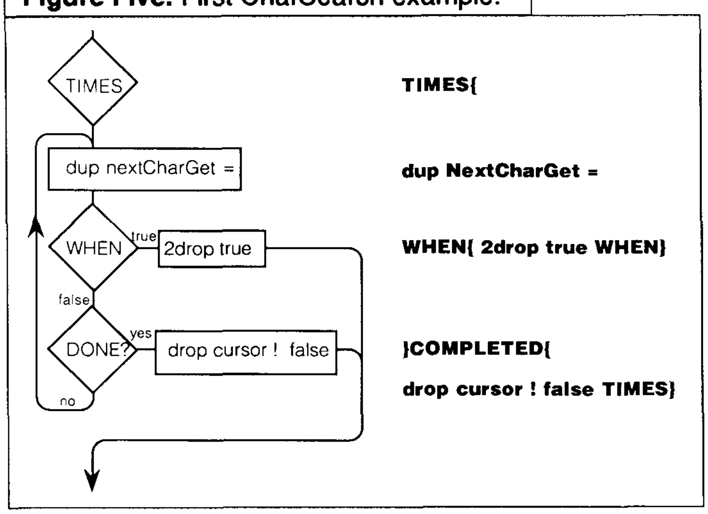
Below is a word to do that, using the new word `}COMPLETED{`.

Figure Four. Return-stack, control-structure frame.




```
: CharSearch ( char -- true | false )
  cursor @ swap  \ keep previous position
  TextEnd @ cursor @ -
    \ # of chars to end of text
  TIMES{ dup ( ... char char )
    NextCharGet  ( char char textChar )
    \ Move the cursor on
    \ while furnishing the
    \ pointed char.
    = WHEN{ 2drop true WHEN}
    }COMPLETED{ drop ( initialCursorPosition)
    Cursor! false
  TIMES}  ;
\ Compare with the flowchart in Figure Five.
```

Figure Five. First CharSearch example.



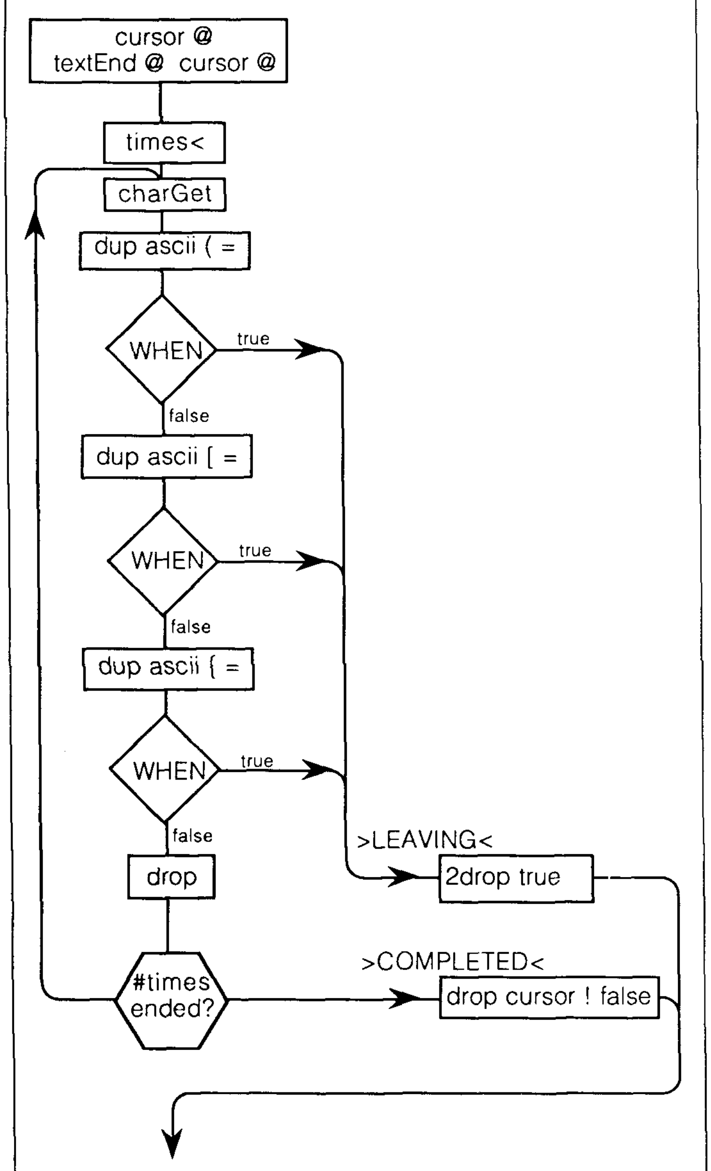
`}LEAVING{` and `}COMPLETED{` can be used together. The next example is a variation of the previous one. Here we want to search our text for the first occurrence of (, [, or {. (See the flowchart in Figure Six.)

```
: CharSearch
  cursor @   \ keep initial cursor position
  textEnd @ cursor @ -
  \ calculate # chars to end of text
  TIMES{ charGet
    dup  ascii ( = when
    dup  ascii [ = when
    dup  ascii { = when
    drop
}COMPLETED{ Cursor! false
  \ Restore initial cursor position.
}LEAVING{ 2drop true
  \ Drop initial cursor position...
    \ ...and character under cursor.
TIMES}  ;
```

LOOP—Discussion and Examples

Standard Forth offers only one kind of definite loop, with two variations: `DO LOOP` and `DO +LOOP`. The `DO LOOP` has

Figure Six. Using }LEAVING{ with }COMPLETED{.



tried, until now, to be both a TIMES loop and a LOOP loop. But when you try to do two things at once, you do them inefficiently. The drawbacks of the DO LOOP are:

- Counter-intuitive position of loop start and end (*end start* instead of *start end*). This has allowed the DO LOOP to work as a TIMES loop when used like 0 DO ... LOOP. (If the positions had been *start end*, we would have had to write 0 SWAP DO ... LOOP.)
- Slow execution when working like TIMES because, instead of decrementing and checking a flag like the TIMES construct, the DO LOOP must increment and check against a limit. (This drawback, coupled with the appearance of Forth processors, has led to the use of FOR NEXT in recent times.)
- Slow execution when working like DO +LOOP because the step is pushed and popped from the stack at every iteration of the loop without any valid reason.

Forth traditionalists could sustain that, in such a manner, it is possible to use a computed loop step that varies from one iteration to the next, but we can observe that real life cases have a constant loop step and, moreover, a much more flexible loop is furnished by the FOR{ ... FOR} construct, which allows for any test at any step (or any new computed index value), and any model of loop pre-increment or post-decrement, etc.

Establishing the Best Input Characteristics for Loops

Definite loops are usually used for doing something a certain number of times, while allowing the index to assume a predefined set of values. The set of values may be specified by three out of the following four parameters.

start	First index value.
end	Last index value (or the value after the last one).
step	Index step, the constant difference between two successive index values.
times	Total number of different values assumed by the index during the loop, equal to the number of times the loop will be executed.

The possible combinations that can be used to specify the set of values the index will assume are:
start end step
start end #times
start #times step
end #times step

(The combination *end #times step* isn't worthy of discussion.)

Start end step is the combination chosen by Forth and other languages. Forth uses loops primarily to work on memory addresses, so *start* specifies the first address we have to work on, and *end* is the limit address. In such cases, it is sometimes useful to specify *start size step* instead, where *size* is the size of memory we want to work on (*size* = *end-start*).

Start end #times can be used when we want to sample a function in an interval given by *start end* with a certain resolution: *#times*. This combination would probably have to be implemented with floating-point numbers—its usefulness with integers is dubious.

Start #times step is used when we work on an array of elements of which we know the starting address, the number of elements, and the element size. This combination is the simplest and most efficient to implement. It will be our base for implementing all other kinds of loops. (We usually know the number of elements in an array and its first memory address.) Moreover, specifying *start #times step* doesn't allow room for misunderstanding.

Specifying *start end step* raises some subtle points to consider. Suppose the specified *end* is *step* aligned with the given *start* (as in 10 start 20 end 2 step). Do we mean that the loop must assume the *end* value, or must it stop at *end - step*? If we want to have the relation between *end, start,* and *size* expressed simply by *size* = *end - start*, we must deduce that the *end* value has to be excluded. Otherwise, the relation between *end, start,* and *size* must be written as *size* = *end - start* + *step*.

Suppose, on the contrary, that the specified *end* isn't *step* aligned with the given *start* (as in 10 start 21 end 2 step).

Is that an error condition? If not, how many times do we have to repeat the loop?

Let's compare the choices of BASIC and Forth, and deduce the relationship between *start, end, step,* and *#times.*

Considering a positive step, in BASIC we write:

```
FOR I=10 TO 18 STEP 3 : PRINT I : NEXT I
  loop is done 3 times
FOR I=10 TO 19 STEP 3 : PRINT I : NEXT I
  loop is done 4 times
FOR I=10 TO 20 STEP 3 : PRINT I : NEXT I
   loop is done 4 times
FOR I=10 TO 21 STEP 3 : PRINT I : NEXT I
   loop is done 4 times
FOR I=10 TO 22 STEP 3 : PRINT I : NEXT I
   loop is done 5 times
```

So the relation is *#times = diff/step+1* where *diff := end - start.*

In standard Forth, a DO +LOOP works like:

```
19 10 DO I .  3 +LOOP  loop is done 3 times
20 10 DO I .  3 +LOOP  loop is done 4 times
21 10 DO I .  3 +LOOP  loop is done 4 times
22 10 DO I .  3 +LOOP  loop is done 4 times
23 10 DO I .  3 +LOOP  loop is done 5 times
```

So the relation is *#times = (diff - 1)/step + 1* (check it against the examples to convince yourself). This relation gets simplified as *#times = diff* when the *step* is 1.

We can observe that the Forth formula giving the *#times* is more complex than the BASIC formula. (If we also consider negative steps, things get much worse for Forth.)

What I consider the simplest choice is to define *#times := diff/step.*** That choice is equivalent to the Forth one for the case *step = 1* (most common). Besides, that choice has some useful consequences when the *step* is not aligned to the *start* (see later examples) while maintaining historical continuity when the *end* is aligned to the *start.*

Additional Support Words for Definite Loops

Keeping the above discussion in mind, let's consider the various ways we can specify an array on which we have to work:

StartAddress #elements SizeOfElements
StartAddress SizeOfArray SizeOfElements
StartAddress LimitAddress SizeOfElements
\ limit address is the first address not belonging to the array
StartAddress LastElementAddress SizeOfElements

So, the generic pattern is *StartAddress ??? SizeOfElements.* Thus, let's define three modifiers SIZE, END, and END] that operate on three numbers and convert them to the standard format (*start, #times, step*):

(Continued on page 37.)

** For subtle-minded readers: The kind of division used—rounded toward negative infinity or toward zero—doesn't bother us except for unusual cases like 10 start 9 end 2 step where, if we round toward negative infinity, the result of the division is negative and so, correctly, the looping construct issues an error message. If we round toward zero, the loop will execute zero times without issuing the error message.

New FIG Board Members

In lieu of the usual "President's Letter," we offer the following statements made by the newest members of the Forth Interest Group's Board of Directors. They were installed last November at the FORML conference, where the new board also held its first meeting. The board is now composed of the following individuals:

John Hall, President
Jack Woehr, Vice-President
Mike Elola, Secretary
Dennis Ruffer, Treasurer
David Petty
Nicholas Solntseff
C.H. Ting

The board welcomes comments from FIG members. John Hall will return in the next issue with his "President's Letter."

Mike Elola

"I started out my involvement in FIG as secretary of the business group that meets once a month to discuss the operation of FIG. I came to my first meeting at the request of Kim Harris, who considered me a good candidate to replace him as secretary. Partly out of respect for him, I agreed to become part of the business group.

"I considered my role as that of an observer for the first couple of years. Soon I overcame my initial skepticism with the business team members and their qualifications. By now, I have gained substantial respect for the leadership skills of the outgoing president, Robert Reiling, as well as the current president, John Hall. (The president presides over the business meetings, and ends up having to referee some very delicate clashes during the long haul.)

"This experience gives me a background with FIG and its leaders, so I feel confident that I can contribute. My biggest concern for FIG has not really changed: I have always been concerned that our collective FIG energies might not be applied properly to obtain needed goals. Along with my help, I now feel that the business group has made considerable progress in setting priorities and focusing its energies. In the years that I have served, I have become especially aware of our limitations. Understanding and confronting those limitations is a vital leadership skill. Otherwise, we can easily squander our limited resources, both in terms of volunteer time and FIG reserves.

"Although our progress at learning to work within our limitations has come slowly, I am proud about the decisions we have made so far. Over the last year, we have spent considerably less money for nearly the same services. More remarkably, we have not sacrificed the quality of those services. (Admittedly, some of the services have been cut. Others besides myself have had to step up our level of volunteer work to compensate.)

"No doubt FIG has persevered because of its determined leaders. I'd like to continue to serve FIG, now more than ever since I expect to be able to enjoy monitoring FIG's financial stabilization, if not recovery. Innovative ideas from all of our business team partners have been essential to help turn things around. Beyond specific measures we have taken, a long soul-searching period has contributed to our success. This has helped instill similar attitudes in most of the business team members, and increases my ability and eagerness to serve.

"Still, we cannot yet rest assured of our future, and I don't know if we ever will (this is not necessarily so bad). I personally feel that any stability we realize by small but steady efforts is more durable than stability or growth that is, perhaps, attainable by occasional concerted efforts involving greater risks.

"I also hope to moderate the efforts of others who would try to vitalize Forth with some kind of slick marketing shtick. Steve Wozniak and the Homebrew Computer Club are now much further away from mainstream culture than they were at one time. A group such as FIG may need to remain in relative obscurity for the foreseeable future. Nevertheless, we should position ourselves comfortably. Acting out of desperation is not the way to inspire and keep the faith of our new and returning members."

Nicholas Solntseff

Dr. Nicholas Solntseff is of Russian emigré background and was born in Shanghai, China well before World War II. He was educated in English in Shanghai and completed his schooling at Sydney Technical High School in Sydney, Australia. Dr. Solntseff attended Sydney University, where he studied physics and obtained his B.Sc. in 1953 and Ph.D. in 1958. After a period of employment in England's nuclear engineering industry, he joined the University of London in 1963. Returning to Australia in 1967, Dr. Solntseff switched to Computer Science and taught at the University of New South Wales until 1970, when he moved to McMaster University (Ontario, Canada) after a year as Visiting Professor at the University of Colorado.

Dr. Solntseff has been involved with Forth since 1981, when he implemented his first fig-Forth on an Ohio Scientific microcomputer. He has been the convenor of the South Ontario Chapter of the Forth Interest Group since its inception early in 1982. Dr. Solntseff's research interests include the implementation of a Forth-like language called Markov, as well as interfacing Forth with Microsoft Windows. For the last two years, Dr. Solntseff and his students have been working on human-interface techniques in Medical Expert Systems being developed in the Department of Clinical Epidemiology, McMaster University.

Jack Woehr

Jack Woehr learned Forth in 1986, and quit his factory

(Continued on page 35.)

Best of GEnie

Gary Smith
Little Rock, Arkansas

Discussion regarding the ANS Forth draft standard continued hot and heavy as we entered 1992. On January 16, the special invited guest in our on-line conference was Greg Bailey of Athena Programming and Technical Subcommittee chair on the X3J14 Technical Committee. Greg's topic was "The Costs and Benefits of Adopting ANS Forth." If you were not present at Greg's guest conference and have not yet captured the transcript, I highly recommend doing so.

There will be no further discussion of the ANS Forth effort in this column in this issue because, by the time you read this, the Technical Committee will have met in mid-February to vote. There may not be that much to discuss or vote on, because as late as mid-January the committee had received zero (that's correct, zilch) comment. If you had specific comments or objections and failed to submit them, you have only yourself to blame. The opportunity was certainly there. I doubt if any standards effort has ever been so open to scrutiny.

So, are there any other hot buttons? You bet! Lots of them. Object-oriented programming and embedded systems still enjoy lively exchanges, but maybe one of the hottest topics on GEnie and ForthNet is minimal Forth kernels that also perform. Witness the topic opened by fellow sysop Elliott Chapin on January 1, 1992. This excerpt, taken on January 17, only runs for 2 1/2 weeks and I already know of at least two outstanding replies.

If you aren't participating in the guest conferences and in these discussions, you are missing out on a lot of the fun associated with being a Forther. Consider joining us soon.

Topic 26: Minimal Forth

How small can a working Forth be? Why try?

The minimal-Forth question has started up again; small wordsets are more than an intellectual exercise. Some processors are very small. Small kernels ease porting.

From: Ralf E. Stranzenbach
Subject: List of Forth words?
Hi,

I'm searching for a list of Forth words that is *required* to be implemented in assembly language to create a reasonable, but very small in size, Forth environment.

Is there anyone who has assembled a list containing those primitives and, possibly, the implementation of the "higher-level words?"
Happy New Year,
—Ralf

From: Milan Merhar
Subject: Minimum Forth environments

Reply to two recent posts:

"I have a (small) 6809 system in order to learn some assembly. Just for fun (!) I want to write a (small) Forth environment. What is about the minimum set of Forth words as a starter. Thus, using this minimum, I write the other words in Forth.
"Regards, Ton 't Lam"

"I'm searching for a list of Forth words that is *required* to be implemented in assembly language to create a reasonable, but very small in size, Forth environment.

"Is there anyone who has assembled a list containing those primitives and, possibly, the implementation of the 'higher-level words'?
"—Ralf"

I've sat in on a couple of informal discussions on this subject. The general consensus is that about ten words are sufficient:

Stack ops:
`DUP` (create a stack element)
`DROP` (destroy a stack element)
`SWAP` (move stack element)
`>R` and `R>` (stack exchange)

Arithmetic/logic:
`LITERAL` (constants, etc.)
`NAND` (sounds silly, but you can synthesize anything else out of it!)

Address-space access:
`!` and `@` (or `C!` and `C@`, if you wish)
`P!` and `P@` (for I/O space port access, if your CPU has such a thing...)

Dictionary extension:
`CREATE`

This is a very sparse list! Even the rawest bootstrap system would probably define a richer set of primitives than this. For example, + and * and / would be a *lot* nicer if they were primitives, rather than colon definitions made of tens or hundreds of primitive ops. Similarly, words like `FIND` are very nice to have!

A more sensible minimum set of primitives may be found in the eForth model; no doubt an implementation of it is available for most any CPU you're interested in. Also look at the current ANS Forth proposal; the Core wordset will give you a good idea as to what functions are needed (although lots of them won't be primitives in an implementation such as you describe).

Discussions of the "angels-on-the-heads-of-pins" variety continue as to which primitives belong on the short list. For example, if you have `R@` you could synthesize `DUP`.

Rob Chapman once proposed a set of primitives for a Forth machine that had two kinds of arithmetic/logical ops; the first kind returns the value of the result, the second kind returns the resulting carry bits.
Regards, Milan J. Merhar

From: Doug Philips
Subject: Looking for a small PD-Forth for the 8086
Ralf E. Stranzenbach writes,

"I've heard about a small Forth named MINI4T41. Does anyone know where to get it?"

I found it in FIG's on-line library on GEnie. It is now available via e-mail from FNEAS. To get it, send a message to:
fneas@willett.pgh.pa.us

with the following body:
send MINI4T41.ARC
path your-email-address-RELATIVE-to-the-INTERNET-goes-here

You *must* supply an Internet relative e-mail address with the path command.
—Doug
Preferred:
dwp@willett.pgh.pa.us
Okay: {pitt,sei}!willett!dwp

From: Nick Janow
Subject: Minimal Forth
Elliot.C writes:

> "The minimal-Forth question has started up again; small wordsets are more than an intellectual exercise. Some processors are very small. Small kernels ease porting."

Minimal kernels might also be valuable on large processors. If the kernel and a program can fit in the cache, it will really scream along.

Nick_Janow@mindlink.bc.ca

From: Andy Valencia
Subject: Looking for a small PD-Forth for the 8086
Doug Philips writes:

> "send MINI4T41.ARC..."

I was disappointed to find that there is no source available for this Forth. If I'm going to live under an opaque execution environment, I usually will go for a richer one, like F-PC. For a spartan environment, I at least want the ability to customize at any level.
Just my opinion...
Andy Valencia

From: RCS
Subject: Haydon's levels of Forth
In Glen Haydon's magnum opus, ***All About Forth*** (3rd edition), his introduction (page ix) describes "levels of Forth":
Level 0: includes the 63 functions Charles Moore has often listed as the basis of Forth. They lack any form of input or output to storage devices.
Level 1: fig with rudimentary storage
Level 2: MVP with a richer function set
Level 3: F83
Level 4: F-PC
Level 5: The future, 32-bit everything

Can someone cite where Moore defined his fundamental 63 functions?
Regards, rcs

From: Ton 't Lam CRC
Subject: Min. Forth *and* good performance
Some time ago I asked for the minimum Forth system. It turned out that nine words are necessary. However, the performance is likely to be lazy. I can imagine. I started with EMIT and KEY, though.
Now as I go along it appeared to be very easy to add new words in assembly. My question now is: What Forth words need to be coded in assembly to have good performance. (My estimation 30 to 50 will do.)
Now I am asking: How is a number (officially) compiled into a word. I.e., how to distinguish a number from an execution address.
In other strings I read the word POSTPONE. What is this? How is it implemented?
—Ton 't Lam

From: Bernd Paysan
Subject: Min. Forth *and* good performance
Ton 't Lam CRC writes:

> "How is a number (officially) compiled into a word. I.e., how to distinguish a number from an execution address."

In basic words:

```
: LIT
  R> DUP CELL+
  >R @ ;

: LITERAL
  POSTPONE LIT ,
  ; IMMEDIATE
```

> "In other strings I read the word POSTPONE. What is this? How is it implemented?"

```
: COMPILE
  R> DUP CELL+
  >R @ , ;

: POSTPONE
  BL WORD FIND
  DUP 0=
  IF <not found code>
    THEN 0<
   IF COMPILE COMPILE
   THEN , ; IMMEDIATE
```

COMPILE is not part of ANS Forth and this definition here is exactly the wrong thing, because it is rather tricky and dependent on a threaded-code Forth. POSTPONE (nice word, awful name) is not that what a Forth programmer does with the things, he doesn't want to do (first postpone them, and then wait...). It postpones the compile time behavior, thus it is COMPILE for non-immediate words, and [COMPILE] for immediate.

> "What Forth words need to be coded in assembly to have a good performance. (My estimation 30 till 50 will do.)"

Arithmetics:
`+ - AND OR XOR CELL+ UM* UM/MOD`

Stack:
`DUP OVER SWAP ROT DROP`

Return-Stack:
`>R R> EXECUTE`

Memory:
`@ ! C@ C! MOVE FILL`

Tests:
`0= 0< < U<`

Dictionary
FIND or an appropriate basic of it

Inner interpreter:
`NEXT DOCOL (DOES>)`

Control flow:
`BRANCH ?BRANCH (LOOP) (+LOOP)`

If you compile words like CELLS (DUP + for 16 bits) by expanding macros, these words are enough to give good performance. It is not very worthwhile to do much more. This is exactly in the range of 30 to 50 words (some more will not [add] much, some less will decrease performance). It may be worth it to add LIT as a code word, since literals are used very often. Some people have tricky ideas about how to realize BRANCH and ?BRANCH, and they are not slow:

```
: BRANCH
  R> DUP @ +
  CELL+ >R ;

: ?BRANCH
  0= R> SWAP OVER
  @ AND + CELL+ >R ;
```

From: Mike Haas
Subject: Min. Forth *and* good performance
Bernd Paysan writes:

> "In basic words:
> ": LIT ...
> ": LITERAL ..."

This did not answer the question. In fact, the question touches on something that is important to me as the author of JForth. JForth includes a standalone-application generator called CLONE. It is used thus:
CLONE <wordname>

This creates an entirely standalone image that includes only the Forth words needed by <wordname>. If <wordname> didn't call EMIT, then EMIT doesn't get CLONEd into the image. This leads to very small executables. (CLONEing NOOP creates a standalone program of about 3K... this is the support necessary to initialize the environment if either double-clicked from an icon or typed into the Amiga's shell.

When CLONE puts together the new image, it has to perform many relocations, since everything is moved around relative to the start of the program (as opposed to the start of the JForth image).

This means that CLONE must be able to tell the difference between a compiled *number* and a compiled *address* (as ['] might produce).

For this purpose, I've implemented ALITERAL.

The concept that a compiled number must be able to be differentiated from a compiled address is not normally needed in Forth, but for sophisticated functionality such as CLONE, it is necessary.

The answer to the above question is that there is no standard way of compiling in an address vs. a literal number...and there should be.

From: Rob Chapman
Subject: Min. Forth *and* good performance

I once had similar high-level definitions for:

```
: BRANCH ( -- )
  R> @ >R ;
: 0BRANCH ( n -- )
  0= R @ CELL -
  R - AND R> +
  CELL + >R ;
```

In this case I got faster branches but slower conditional branches.

0BRANCH and 0= represent the classic chicken-and-egg syndrome. In the above definition, we depend on 0= as a primitive. However, if we wish to define 0= in Forth:

```
: 0= ( n -- f )
  IF -1 ELSE
  0 THEN ;
```

then we need conditional branching as a primitive.

Other amusements dredged up from the cellar [are given in Figure One.]
—Rob

From: E.RATHER [Elizabeth]
Horrors!

From: Mitch Bradley
Subject: Min. Forth *and* good performance
Mike Haas writes,

"When [JForth's] CLONE puts together the new image, it has to perform many relocations, since everything is moved around relative to the start of the program (as opposed to the start of the JForth image).

"This means that CLONE must be able to tell the difference between a compiled *number* and a compiled *address* (as ['] might produce).

"For this purpose, I've implemented ALITERAL."

My Sun Forth and Forthmacs systems, which are fully relocatable and support an "application stripper" program that does the same thing as JForth's CLONE, have the same problem and solve it in the same way. My equivalent of ALITERAL is named (').

"The answer to the above question is that there is no standard way of compiling in an address vs. a literal number... and there should be."

You can do it in ANS Forth by using POSTPONE with ['] or by using something like:
S" ['] FOO" EVALUATE

possibly with a string that is constructed at run time.
Mitch.Bradley@Eng.Sun.COM

Figure One.

```
: EXECUTE  ( tick -- )  >R ;  ( not that portable though! )
: ROT  ( a \ b \ c -- b \ c \ a )  >R SWAP  R> SWAP ;

( ==== Inner Interpreters ==== )
: (VAR)  ( -- addr )  R> ;
: (CONST)  ( -- n )  R> @ ;
: (NEXT)  ( -- )  R>  R> ?DUP  IF  1 - >R  @  ELSE  CELL +  ENDIF  >R ;
: LIT  ( -- n )  R> @+ >R ;
: (DO)  ( limit \ index -- )  SWAP  R>  SWAP >R  SWAP >R  >R ;
: (LOOP)  ( -- )  R>  R> 1 + DUP R <
   IF  >R  DUP @ +  ELSE  R> 2DROP  CELL +  ENDIF  >R ;
: (+LOOP)  ( n -- )  R> SWAP  DUP R> +  SWAP 0<  OVER R <  XOR
   IF >R  DUP @ +  ELSE  R> 2DROP  CELL +  ENDIF  >R ;

( ==== Comparisons via divide and conquer ==== )
: <   ( n \ m -- flag )  2DUP XOR 0<  IF  DROP  ELSE  -       ENDIF  0< ;
: >   ( n \ m -- flag )  2DUP XOR 0<  IF  NIP   ELSE  SWAP -  ENDIF  0< ;
: U<  ( n \ m -- flag )  2DUP XOR 0<  IF  NIP   ELSE  -       ENDIF  0< ;
: U>  ( n \ m -- flag )  2DUP XOR 0<  IF  DROP  ELSE  SWAP -  ENDIF  0< ;
```

For the really esoteric:

```
( ==== Unsigned multiplication and division ==== )
: quot<  ( n \ q -- q )  2* OR ;
: rem<m ( r \ m -- r )  0< 1 AND  SWAP 2*  OR ;
: div?  ( n \ r -- n \ ?rn- \ f )  OVER -  DUP 0<
   IF  OVER +  0  ELSE  1  ENDIF ;
: /MOD  ( m \ n -- r \ q )  SWAP  0 OVER  rem<m  SWAP 2*  ( n \ r \ m/q )
   F  FOR  >R  div?  SWAP  R rem<m  SWAP R> quot<  NEXT
   >R  div? >R  NIP  R>  R> quot< ;
: /  ( n \ m -- quot )  /MOD NIP ;
: MOD  ( n \ m -- rem )  /MOD DROP ;

: *  ( n \ m -- nm* )  ( unsigned )  0 SWAP
   F  FOR  DUP >R  0<  IF  OVER +  ENDIF  2*  R> 2*  NEXT
   0<  IF  +  ELSE  NIP  ENDIF ;
```

(Continued from page 31.)

job to become a programmer. He has been steadily employed in Forth since that time. He currently chairs the Software Collegium at Vesta Technologies, where he has been employed since February 1988. (Vesta manufactures single-board computers for embedded control with Forth in ROM.) Besides serving as the Vice President of FIG, Jack is a Contributing Editor for *Embedded Systems Programming* magazine, the author of *Seeing Forth* (Offete Enterprises, 1992), frequently writes Forth articles for *Dr. Dobb's Journal*, and is the author of JAX4TH (the first dpANS-Forth for the Amiga).

"The Forth community is a besieged minority. The Forth Interest Group has suffered a decline in recent years. Old members have left the group, and new members are slow to replace them. At the same time, Forth professionals are confronted with the paradox of Forth usage increasing while Forth market share is declining.

"These are ominous runes cast at the feet of Forth.

"Yet we believe that our heterodox model of computation presents a more holistic approach to the interaction of man and machine than the path taken by orthodox computer scientists. If our approach still has value, then the practices and institutions which have served us are worth the effort taken to maintain them.

"I have sought and accepted admission to the Board of Directors of the Forth Interest Group in order to contribute to the preservation of an organization which has proved so useful in the past decade, an organization which hopefully shall continue to render effective service in the coming decade.

"The Forth Interest Group has historically fulfilled two important roles, that of aiding newcomers entering upon the path to Forth proficiency, and that of a mutual aid society for Forth programmers. I know this from experience. I learned Forth in the course of many entertaining Saturdays spent at Wolf and Pruneridge Roads. Furthermore, each phase of my successful career in Forth has involved employment found either at a meeting of the Forth Interest Group or via the GEnie Forth Interest Group RoundTable.

"I call upon all enthusiastic exponents of the Forth approach to urge their Forth acquaintances to join or to renew their membership in the Forth Interest Group. FIG will try to keep up its end of the bargain by constantly improving the quality of *Forth Dimensions* and by increased attention to the needs of beginners, to local chapters, and to community activities for the promotion and benefit of Forth.

"It's time to vote with your feet: if you walk away from FIG, FIG will become merely a part of computer club history like the Homebrew society. If you stay, and if you bring others, FIG will continue as your tutor, friend, and advocate. Now it's up to you."

—Jack Woehr
jax@well.UUCP
JAX on GEnie
Sysop, RCFB 303-278-0364
FAX: 303-422-9800

If you stay, and if you bring others, FIG will continue as your tutor, friend, and advocate.

$ New Contest Announcement $

Call for Papers!

Forth Dimensions is sponsoring a contest for articles about

"Forth on a Grand Scale"

Write about large-scale Forth applications, systems, or ...

1st prize: $500
2nd prize: $250
3rd prize: $100

See editorial in this issue.

Volume XII Index

A subject index to Forth Dimensions *contents published from May '90 – April '91. Prepared by Mike Elola.*

(Continued from page 30.)

```
: SIZE ( StartAddr ArraySize ElementSize --
                          -- Start #times step )
  dup >R / R>  ;

: END  ( StartAddr LimitAddr ElementSize --
                          -- Start #times step )
  >R OVER - R@ / R>  ;

: END]  ( StartAddr LastAddr ElementSize --
                          -- start #times step )
  END swap 1+ swap  ;
  \ this last word isn't really felt useful
```

Moreover, sometimes an array must be scanned in reverse order, although it's easier to specify the array by its starting address. Thus, let's define the additional modifier BACK to be used like

```
BACK LOOP{ ... LOOP}
```

This has the effect of reversing the order of the values assumed by the index in the absence of BACK.

```
: BACK  (StartAddr #elements ElementSize --
       -- LastAddr #elements ElementSize )
  DUP NEGATE >R
  OVER 1- *
  ROT + SWAP
  R>  ;
```

With the above choices and definitions, we have a very flexible loop construct that accepts various input formats and greatly simplifies work with arrays.

Obviously, depending on the problem at hand, similar techniques may be used to extend the patterns accepted by the loop construct, making it possible to feed the construct with a format natural to the problem at hand. Moreover, the basic format is very easy to implement, fast to execute, and precise in meaning.

Examples

Create and initialize a table:

```
100 CONSTANT #LOGOS
29 CONSTANT SIMPLELOGO
CREATE LOGOS  #LOGOS CELLS ALLOT

: LOGOSINIT
  LOGOS #LOGOS CELL
  LOOP{
    SIMPLELOGO I !
  LOOP} ;
```

Search our table for a specified value and leave its address if it is found:

```
: logoSearch ( logo -- false | addr true )
  LOGOS #LOGOS CELL
  LOOP{
    dup I @ =
    WHEN{
      drop I true
    WHEN}
  }COMPLETED{ drop false
  LOOP}  ;
```

Search in reverse order:

```
: logoBackSearch ( logo -- false | addr true)
  LOGOS #LOGOS CELL BACK
  LOOP{
    dup I @ =
    WHEN{
      drop I true
    WHEN}
  }COMPLETED{ drop false
  LOOP}  ;
```

(Code, figures, and article continue in next issue.)

ADVERTISERS INDEX

A FORML Thanksgiving

Richard Molen
Huntington Beach, CA

On November 25, 1991, just over forty dedicated Forthers flocked to Asilomar on California's Monterey peninsula to participate in the FORML conference. Some went to exchange ideas, some went to exchange addresses, all went to exchange experiences and to increase the collective knowledge base of the Forth community. This year there was something for just about everyone.

As we arrived at Asilomar, the air was cool and clear with a strong breeze. As we rushed in (late) to register and pick up our name badges and notebooks, we were intercepted with a big hug by Wil Baden who makes it a point to greet everyone at FORML in this fashion. It's a nice way to start.

He supplied the code and theory for creating and killing simulated organisms...

Asilomar is a beautiful retreat sprinkled with windswept Monterey pine and cypress trees. Weathered boardwalks cut through the struggling vegetation in the dune restoration project, leading to a beach of white sand and deep blue water—a living picture. Early morning finds deer foraging among the trees and a seascape full of life. In these surroundings, it was easy to relax and concentrate on the conference itself. As soon as we were registered, the lunch bell rang and we went to the cafeteria to eat. I was impressed by the simple elegance of the cafeteria. It wasn't until after we were seated that I noticed there weren't any menus. How nice it was to not worry about what to eat. "Could there be a GUI lesson here?" I mused. I enjoyed my French dip sandwich, as Guy Grotke gave me bits of information to mentally munch on. He talked about the three-dimensional mouse project he has been working on and the ADSP2105 chip that costs less than $10.00 apiece.

Adjourning from lunch, we assembled in Merrill Hall, a large rustic building on top of a small knoll. Robert Reiling began the conference by welcoming us and announcing the agenda. Mike Perry moderated. Having enjoyed a good lunch, I looked forward to the veritable smorgasbord of knowledge about to be served.

The first presentation came from Mike Elola. His eyes lit up as he described HomeComing Forth, his implementation of a minimal Forth system on top of Apple's HyperCard environment. HomeComing Forth is simple, with very nice debugging tools. For example, a definition's object and source code are displayed side-by-side when editing, encouraging users to see what is actually compiled. Mike's paper was quite tasty.

If you have ever crashed your system by leaving an unbalanced return stack, raise your hand, lower your head, or at least read on. Roland Koluvek's paper describes and implements return stack security using a temporary stack for compile-time housekeeping. While this takes a little more compile time, it is well worth it for applications which allow users open access to the Forth environment. In addition, he added a prompt which displays the top three cells on the temporary stack when compiling a definition interactively. This allows a programmer to see what's happening as the definition is compiling. Roland received the "most in keeping with FORML" award (a bottle of wine) for this effort.

Guy Kelly served up a very informative entree by speaking of his efforts to characterize tradeoffs in various Forth architectures. In his paper, he benchmarked and characterized 19 of the most common Forths. *[See article in this issue.—Ed.]* He also documented the sequences required to open a file and load a program, the various assembler syntaxes, and a brief on each of these Forths. In addition, he further isolated the effects of threading, segmenting, and register usage by manipulating each of these components using riFORTH as a base system. Guy found only a 2:1 performance ratio between the fastest and slowest of these versions. He concluded that other considerations often outweigh this performance gain. This certainly surprised me. His tests and riFORTH are available on GEnie.

One of my favorite foods for thought is metacompilation. Guy dispels the mysteries of metacompilation and offers the metacompiler which he used for the benchmarking project. This metacompiler, also available on GEnie, is capable of generating new Forth systems with various threading and memory-segmenting schemes. Guy's papers are a must for anyone who wants to experiment with Forth architectures. It is easy to see why Guy Kelly won the "Public Service" award. Thank you, Guy.

Andrew McKewan spiced up the Motorola 6805 emulator with an optimizing Forth native-code compiler. This must be ambrosia for anyone working with the 6805 emulator. He commented that after he tossed out the

PCYerk Classes

Rick Grehan
Peterborough, New Hampshire

Rick Grehan is a senior editor at BYTE magazine and the technical director of BYTE Lab. He first encountered Forth over seven years ago when developing a music synthesizer control system built around a KIM-1. Since then, he has used Forth on 68000 systems (including the Macintosh), the Apple II, and the IBM PC. He has also done extensive work on the SC32 stack-based processor. Rick has a B.S. degree in physics and applied mathematics, and an M.S. degree in mathematics/computer science. His work on a PC version of the Yerk implementation won first prize in *FD*'s object-oriented Forth contest.

The following code builds on the object-oriented Forth discussed in the last issue of *FD.*

```
\ *******************
\ ** BASIC CLASSES **
\ *******************

\ *********
\ ** object
\ *********
:class      object
     0      ivar  dummy        \ Used to get offset to ivar area

\ Return address to object's instantiaion in variable
\ segment.  You can use this to get an object's address
\ and store it in a variable for deferred binding.
\ E.G.:
\    variable frank
\    12 word_array bob
\    44 fill: bob
\    addr: bob frank !
\    2 get: { frank @ } .   FORTH RESPONDS>>  44 ok
\
:m addr:          ( -- addr )
   dummy 2-
;m

\ Return address to start of object's ivars region
:m ivar-addr:     ( -- addr )
   dummy
;m

\ Return the length of the object's data area
:m length:        ( -- n )
   dummy 2-                   \ Get pointer to instantiation
   @                          \ Class address in token segment
   @t                         \ Length
;m
;class
```

(FORML, continued.)

idea that Forth had to be 16 bits, indirect threaded, and interpretive, he was able to make this 8-bit native code system. This useful insight applies to applications as well.

Frank Sergeant has written a three-instruction Forth for embedded system development on a budget. I don't have room here to elaborate in detail—well, only three words, I guess I do. The only words needed to start developing on an embedded system are `X@`, `X!`, and `XCALL` which fetch, store, and execute a routine on the target system, respectively. Frank described how he implemented these words in an MC68HC11 chip. Having used a Cadillac, four-word variation, I'd have to say that Frank is right on target.

Anyone who has done serious development can appreciate the usefulness of version control and file comparison. Wil Baden presented his tools, which he has ported many times, over many systems, over many years. Wil distributed 20 pages of code forming the basis of a text-file-based, source code control system. His implementation is capable of comparing and collating large files, and keeps all versions of a file in a compact format. It is a useful tool in any language. I found it interesting that some of those who used blocks did not see a need for such tools. Perhaps the modularity of blocks, combined with the fact that the majority of those using blocks used date stamping, reduces the need for such tools. Those interested in receiving a copy of this code on disk should contact Wil.

Guy Kelly called our attention to some of the tradeoffs of interpreting source from text files. By adding some intelligence to parsing words, Guy simplified the definitions of words which use them (i.e., (, . (, .", \, and LOAD). By using block buffers, he eliminated the need for extra text-file line buffers, further simplifying the system. His system, also on GEnie, is simpler and more capable than a BASIS 17 system would be.

On the educational front, Dr. Tim Hendtlass of the Physics Department at the Swinburne Institute of Technology in Hawthorn, Australia, gave an excellent testimonial to the power of Forth in education. Tim described in detail the challenges of teaching interfacing (hardware) to classes of 60 students with various unrelated backgrounds and the dramatic change in their ability to learn interfacing when Forth was used. His paper also contains the exercises used to teach these students to solve simple instrumentation problems using both interrupts and multitasking. People learning to write interrupt service routines can really benefit from Tim's paper. Software for this paper is available on GEnie.

AI buffs and elderly people should take note of Dr. Hendtlass's paper on the development of a distributed, intelligent system which he calls Embedded Node Collectives. Each node collects information, uses an expert system and, sometimes, a neural network to digest this information, and communicates with the outside world in some fashion. These nodes have been used in several systems. The system he cites is one which helps elderly people care for themselves. The neural network—an input into the expert system—

```
\ ********************
\ ** STORAGE CLASSES **
\ ********************

\ ********************************
\ ** 1darray -- 1 dimensional array
\ ********************************
:class 1darray    <super object
      2     ivar  nelems       \ # of elements in the array
      2     ivar  elemsize     \ Size of each element in bytes

\ Allocate space for the array.
:m allocate:        ( n -- )
   dup nelems !                \ Store # of elements
   elemsize @ * allot          \ Set aside space in vars segment
;m

\ Set the elements size
:m setsize:         ( n -- )
   elemsize !
;m

\ Initialize the array.
\ n is # of elements in the array
\ m is the element size
:m init:     ( n m -- )
   setsize: self        \ Set the element size
   allocate: self       \ Allocate memory
;m

\ Return the # of elements
:m #elems:          ( -- n )
   nelems @
;m

\ Return length of data area
:m length:          ( -- n )
   length: self                \ Header information
   #elems: self
   elemsize @ *                \ Length of data portion
   +                           \ Add it all
;m

\ Do bounds checking for index
:m idx-check:             ( i -- i )
   dup 1+ nelems @ >          \ Check bounds
   if clear-o&mstacks         \ Clear the stacks
           abort" Array bounds exceeded"
   endif
;m

\ Return the address of the array members start
:m array_addr:            ( -- addr )
   elemsize 2+
;m
;class
```

```
\ *************
\ ** byte_array
\ *************
:class byte_array <super 1darray

\ Initialize the array
:m init:     ( n -- )
   1 setsize: self             \ Set the element size
   allocate: self              \ Allocate space
;m

\ Return value at index location
:m get:              ( i -- val )
   idx-check: self             \ Check bounds
   array_addr: self            \ Start of array
   +                           \ Add index
   c@                          \ Fetch
;m

\ Set value at index location
:m put:              ( val i -- )
   idx-check: self             \ Check bounds
   array_addr: self            \ Start of array
   +                           \ Add index
   c!                          \ Store
;m

\ Fill the array with value
:m fill:     ( val -- )
   array_addr: self            \ Get address
   #elems: self                \ # of elements
   rot fill                    \ Do it
;m

\ Clear the array
:m clear:    ( -- )
   0 fill: self
;m

init: <<init-method            \ Set initialization method
;class

\ *************
\ ** word_array
\ *************
:class word_array <super byte_array

\ Initialize the array
:m init:     ( n -- )
   2 setsize: self             \ Set the element size
   allocate: self              \ Allocate space
;m

\ Return value at index location
:m get:              ( i -- val )
   idx-check: self             \ Check bounds
   2*                          \ Index -> offset
   array_addr: self            \ Start of array
   +                           \ Add index
   @                           \ Fetch
```

learns a person's habits. The expert system evaluates its inputs to determine what action is needed: a gentle prompting, a phone call for help, etc. I don't think George Orwell would have cared much for this system, but it certainly can be instrumental in helping an elderly person to be more self sufficient. Thank you, Tim, not just for your presentation and software, but also for your recent neural network articles in *Forth Dimensions*.

Plenty of treats were to be had for the hardwired Forthers. Brad Rodriguez discussed his PISC-1 (Pathetic Instruction Set Computer), which uses 1976 TTL technology, has a mere 2100 gates, and implements Forth in microcode. PISC-1 adds a whole new dimension to the phrase "lean and mean."

Dr. Ting showed us how we can create our own chips at the kitchen table by using the National Security Agency's public-domain CMOSN macro cell library. So where's the DIP? Well, Dr. Ting showed us that, too, by using the library to create a 40-pin Data Comparator Chip. With plenty of hand waving, which he promised us in his paper, John Rible discussed his QS2 (QuickSand 2) project proposal for a graduate-level VLSI design project at the University of California at Santa Cruz. It is a 16-bit microprocessor with classical RISC features, which has, among other things, a hardware-based threaded-code interpreter.

Chuck Moore demonstrated his MuP20 chip emulator software which displayed each layer of the chip in a different color. Using the seven-button interface, he scrolled through the chip, displaying layers both individually and combined. Chuck pointed out that, since the chip can be emulated, the circuitry is tested and the making of the chip is anticlimactic. What caught my eye was the simplicity of the user interface. It seemed so simple I wondered if my four-year-old daughter could learn to use it. The emulator did what he needed—no more, no less. Trivial decisions which could distract him (or any user) from his thinking were all but eliminated. Thank you, Chuck.

DR. TING gave the recipe for primordial soup by specifying the modules needed to implement the Tieara Computer Organism System. He also supplied the code and theory for creating and killing simulated organisms, with a challenge to add the mutation-and-evolution components.

JEFF FOX won the "Programming Virtual Hardware" award with his simulation (in F-PC) of the MuP20 running eFORTH. As if running a simulation of CHUCK MOORE's latest chip wouldn't be enough fun, Jeff also simulated (with eFORTH and DesqView) parallel processing with the F20, an enhanced MuP20, using FORTH-Linda—a bulletin board style parallel-processor manager. This must mean that Jeff is simulating virtual machines based on the simulation of a virtual processor, which runs on a virtual machine (i.e., eForth and DesqView). Jeff's paper describes his efforts in detail.

Another tasty dish was DR. TING's talk about the Catalyst, his contribution to the Human Genome Project—the greatest reverse engineering project of all time. The Catalyst is an auto-

```
;m

\ Set value at index location
:m put:           ( val i -- )
   idx-check: self            \ Check bounds
   2*                         \ Index -> offset
   array_addr: self           \ Start of array
   +                          \ Add index
   !                          \ Store
;m

\ Fill the array with value
:m fill:     ( val -- )
   array_addr: self           \ Get address
   #elems: self               \ # of elements
   0 do
            2dup i 2* +       \ Form address
            !                 \ Store value
   loop
   2drop                      \ Clear stack
;m

init: <<init-method
;class

\ *********
\ ** String
\ *********
\ A string object consists of a maximum byte count, byte
\ count, and trailing null byte.  The maximum count does
\ NOT include the preceding byte count and trailing null
\ byte.
:class string <super object
      1     ivar  maxcount
      0     ivar  thestring

\ Allocate space for the string
\ n is # of bytes to allocate
:m allocate       ( n -- )
   dup maxcount c!            \ Save in max. count
   2+                         \ For byte count & null byte
   allot
;m

\ Clear the string
:m clear:
   0 thestring !
;m

\ Store a string in the string object
\ addr must point to a packed, null-terminated string
:m put:           ( addr -- )
   \ See if the string will fit
   dup c@ maxcount
   >
   if clear-o&mstacks
            abort" String too long"
   endif
   thestring $!
;m
```

```
\ Copy contents of string object to destination address
\ Note usage of Upper Deck Forth's $! operator
:m get:            ( addr -- )
   thestring swap $!
;m

\ Return the address of the first character of the
\ string and the byte count
:m count:    ( -- addr n )
   thestring count
;m

allocate: <<init-method

;class

\ ***************
\ ** String array
\ ***************
\ This array is a collection of pointers to string
\ elements.
:class string_array <super word_array

\ Allocate space for the array
\ n = # of elements
\ m = max. size for each string element.
:m allocate:      ( m n -- )
   dup init: self              \ Make space for it
   ['] string >body @          \ Need this to make string
   swap
   0 do
      here >r                  \ Save location
      2dup instantiate         \ Make a string object
      r> i put: self           \ Store addr of object
   loop
   2drop                       \ Clear stack
;m

\ Fetch a string at index i.  Store in address addr
\ Lots of "get:" messages here.  The one in the curly
\ brackets goes to an integer array, and fetches the address
\ of a string object.  The get outside the curly brackets
\ sends a get to a string type.
:m get:           ( i addr -- )
   swap
   get: { get: self }          \ Fetch
;m

\ Place a string at element in index i.
:m put:           ( addr i -- )
   put: { get: self }
;m

allocate: <<init-method

;class
```

mated molecular biology workstation designed to automate the HGP bottleneck of preparing DNA fragments for analysis. At its core is a three-axes robot arm capable of delivering liquids to .001" accuracy. The software runs on a Macintosh and was written in polyFORTH using a simple round-robin tasker.

Dennis Ruffer spoke last. He pointed out the need for a common validation suite for testing whether a Forth system is compliant. His paper discusses the labeling and documentation aspects of this effort. Dennis is looking for people willing to work with him to develop the suite. In my opinion, a common test suite would force interpretation of the standard in areas where it is unclear and, in a sense, test the validity of the standard itself. Dennis can be contacted at Forth, Inc.

I could talk about the wine and cheese parties (which were fun), the impromptu talks (there were some gems), and the workshops (which were lively) that are the less formal parts of FORML, but I'm running out of room. The presentations were wonderful and I look forward to reviewing many of them in detail, but what I found to be at least as inspiring were the people themselves—their experience, their personalities, and their insights. The presentations will be published shortly in the conference proceedings, but this dimension of FORML will only be captured in the minds of those that attended.

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

Second Class
Postage Paid at
San Jose, CA