SILICON COMPOSERS

INTRODUCES THE

SC/FOX™ DEVELOPMENT SYSTEM

SC/FOX COPROCESSOR
Forth Optimized eXpress™

USING HARRIS RTX 2000™ FORTH CPU

- Full-length PC, XT, AT, 386 plug-in board
- 10 MIPS operation at 10 MHz
- 8 MHz standard, 10 MHz upgrade
- Zero-wait state 45/35 ns static RAM
- 64K byte standard, 1M byte upgrade
- All memory accessible by host PC
- Runs in parallel with host PC
- Multiple SC/FOXs in single host PC
- Expand hardware via two terminal strips
- Includes FCompiler™ optimizing Forth
- 8 MHz 64K SC/FOX Unit Price: $1,995
- Delivery: 2 weeks ARO

- RTX Real Time Express™ CPU:
  1-cycle 16X16=32 bit multiply
  1-cycle 14-priority interrupts
  two 256-word stack memories
  three 16-bit timer/counters
  8-channel 16-bit I/O bus

- Optional SC/FOX Products:
  SC/Forth™ Forth-83 Standard Forth
  XRUN™ eXtended RUN Utility
  SC/Float™ floating point library
  SC/FOX/SP™ Serial/Parallel Board
  SC/PROTO™ Prototype Board

Ideal for real-time control, data acquisition and reduction, image or signal processing, or computation intense applications. For additional information, please contact us at:

SILICON COMPOSERS INC.
210 California Ave., Suite K, Palo Alto, CA 94306  (415) 322-8763
A CONVENIENT EXTRA STACK - VICTOR H. YNGVE

How does a stack work? Here is a way to define an extra, general-purpose stack — a textbook example that is also a useful utility. With operators analogous to those for the parameter and return stacks, this extra stack is easy to use instead of the return stack for temporary storage, for input that would otherwise complicate parameter-stack operations, and for other purposes that we hope readers will share in upcoming issues.

A SHADOW STACK FOR DOUBLE NUMBERS - DARREL JOHANSEN

Finally, a way to eliminate mixed-precision operators from Forth code. This "shadow stack" unobtrusively saves the high 16 bits of your numeric entries, and silently surrenders them when 32-bit arguments are required. Think about this: fewer operators, simpler stack operations, and an intriguing avenue to code compatibility between systems.

WISC AND THE FORTH DILEMMA - GLEN B. HAYDON

How does a maker of Forth-based systems provide the full functionality of contemporary programming and operating environments without compromising Forth's simplicity, elegance, and intimate relation to the hardware? This apparent dilemma brings to light many of the important decisions — answered and unanswered — that face today's developers.

USING A STRING STACK - RON BRAITHWAITE

Traditional Forth techniques for working with strings are admittedly limited. This paper presents a nicely rounded implementation based on string-manipulation principles found in the MUMPS computer language. It features a dedicated stack and a complete vocabulary that includes pattern matching. Guess what else — it handily outperforms its progenitor. (Code continued in next issue.)

ABOUT F83's WORDS - TIMOTHY HUANG

What happens when you execute WORDS? F83's vocabulary list buffets our biological buffers with more information than called for by most occasions and most users. A different implementation of the WORDS mechanism allows the original function, or an optional display of selected keywords.

DESIGNING DATA STRUCTURES - MIKE ELOLA

Object-oriented programming and data abstraction make data structures easier to port. Forth applications use such techniques, even if Forth itself remains essentially unchanged. Like factoring, object programming can be implemented as a design philosophy rather than as an imposition of a foreign syntax.

Editorial

Best of GEnie

Advertisers Index

FIG Chapters
EDITORIAL

Stacks of stacks... Our pages usually tend more toward the eclectic than to the thematically ordered, but this issue is an exception. The trio of stacks presented here may inspire you to rethink two of Forth's fundamental characteristics: explicit control of stack operations and the ability to recompile Forth. These authors remind us that special-purpose stacks can be created as easily as other routines. Study these ideas and exercise the code; add some backspin of your own and, of course, let us know what happens!

Our best wishes go to Ron Braithwaite and his wife, Liz, on their recent marriage. Ron is a long-time inhabitant of the Forth community who found a challenge in Forth's lack of uniform (or any) string operators. His comprehensive solution couples a string stack with operators based on the noteworthy string features found in MUMPS. The code for this package is a bit lengthy for our format, so about half is presented here, with the rest following in the next issue. Those of you who get on-line with the Forth Interest Group's GEnie RoundTable can download the code from its software library.

I've been thinking of the Forth programmer's relationship to hardware and to the art of problem solving. He is unrestrained from exploration and trial implementation, in a system which accommodates the oddest whims with minimal penalty. Like the driver of a fine sports car, he is aligned with the working hardware and can wring out its best performance. Because the Forth virtual machine is so closely attuned to the physical architecture, the programmer can "feel the pavement," and judge the balance between finesse and power. Which, as Mahlon Kelly points out ("Best of GEnie," this issue), makes Forth the ideal adjunct to computer science classes.

The short-term trend in microcomputing seems to favor power and control over elegance and intimacy. Like the much-maligned male of the eighties, we must find the harmonious relationship of these qualities. Increasingly complex operating systems and interfaces barricade many systems with protocols and black boxes, while even the big Apple presents Unix "for the rest of us." We are drawn into designs conceived by committee, implemented in pieces too removed from both the problem and the solution, and spot-welded into place by other teams working under management whose chief task is to maintain organizational dynamics and to fight entropy.

In the Forth world, there will always be opportunities for a single person to make a significant contribution. It is an arena for the programmer who remembers that, when microcomputers were developed, it was about much more than squeezing a mainframe into a smaller box — it was about personal freedom.

—Marlin Ouverson
Editor

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

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

Copyright © 1988 by Forth Interest Group, Inc. The material contained in this periodical (but not the code) is copyrighted by the individual authors of the articles and by Forth Interest Group, Inc., respectively. Any reproduction or use of this periodical as it is compiled or the articles, except reproductions for non-commercial purposes, without the written permission of Forth Interest Group, Inc. is a violation of the Copyright Laws. Any code bearing a copyright notice, however, can be used only with permission of the copyright holder.

About the Forth Interest Group

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

"Forth Dimensions" is published bimonthly for $24/36 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to Forth Dimensions, P.O. Box 8231, San Jose, CA 95155."
Here is a little confection, an extra stack that adds features of convenience to a textbook example, making it into a useful programming utility.

Using this extra stack is simple. In analogy to the return-stack words >R, R>, and R@, one writes >X, X>, and X@. Several additional words are provided also. XCLEAR clears the extra stack. This is needed because, unlike the parameter stack and the return stack, the extra stack is not automatically cleared or reset after an error. In analogy to the parameter-stack words PICK and DEPTH, one can use XPICK and XDEPTH. Of course, 0 XPICK yields the same results as X@, and n XPICK copies the nth item from the extra stack to the top of the parameter stack. For added convenience, .X is provided to dump the contents of the extra stack.

There are many uses for an extra stack. It can be used for temporary storage in place of the return stack in cases where >R and R> cannot be used because the return stack would be left unbalanced, or because there would be interference with loop limits and indices, or with words that expect to find a return on the return stack.

For complex definitions with several input parameters, some can be moved to the extra stack and retrieved as needed, thus simplifying the definition by reducing the number of items that have to be juggled on the parameter stack.

Alternatively, some words can expect to find their input parameters on the extra stack, and could leave their results there, thus reserving the parameter stack as an internal calculation stack.

An extra stack has advantages over variables for these purposes. It is reusable by several nested words without interference and it does not, like separate variables, need separate names.

The size of the extra stack is determined by the constant XSSIZE on screen six. This is shown set to three as an aid in checking out the stack words. After checkout, it should be set to the size needed for the application.

This stack works with a stack pointer rather than with a stack address and offset. The first cell in the stack array XSTACK contains the stack pointer, which is the address of the top item on the stack rather than the first free cell, as is sometimes the case with stack pointers. Thus, the code for >X first increments the stack pointer by two (Continued on page 10.)
HS/FORTH

POWER

HS/FORTH's compilation and execution speeds are unsurpassed. Compiling at 20,000 lines per minute, it compiles faster than many systems link. For real jobs, execution speed is unsurpassed as well. Even non-optimized programs run as fast as ones produced by most C compilers. Forth systems designed to look benchmarks are slightly faster on nearly empty do loops, but bog down when the colon nesting level approaches anything useful, and have much greater memory overhead for each definition. Our optimizer gives assembler language performance even for deeply nested definitions containing complex data and control structures.

HS/FORTH provides the best architecture, so good that another major vendor "cloned" (rather poorly) many of its features. Our Forth uses all available memory for both programs and data with almost no execution time penalty, and very little memory overhead. None at all for programs smaller than 200KB. And you can rezone segments anytime, without a system regen. With the GigaForth option, your programs transparently enter native mode and expand into 16 Meg extended memory.

Benefits beyond speed and program size include word redefinition at any time and vocabulary structures that can be changed at will, for instance from simple to hashed, or from 79 Standard to Forth 83. You can behead word names and reclaim space at any time. This includes automatic removal of a colon definition's local variables.

Colon definitions can execute inside machine code primitives, great for interrupt & exception handlers. Multi-cfa words are easily implemented. And code words become incredibly powerful, with multiple entry points not requiring jumps over word fragments. One of many reasons our system is much more compact than its immense dictionary (1600 words) would imply.

INCREDIBLE FLEXIBILITY

The Rosetta Stone Dynamic Linker opens the world of utility libraries. Link to resident routines or link & remove routines interactively. HS/FORTH preserves relocatability of loaded libraries. Link to BTRIEVE METAWINDOWS HALO HOOPS ad infinitum. Our call and data structure words provide easy linkage.

HS/FORTH runs both 79 Standard and Forth 83 programs, and has extensions covering vocabulary search order and the complete Forth 83 test suite. It loads and runs all FIG Libraries, the main difference being they load and run faster, and you can develop larger applications than with any other system. We like source code in text files, but support both file and sector mapped Forth block interfaces. Both line and block file loading can be needed to any depth and includes automatic path search.

FUNCTIONALITY

More important than how fast a system executes, is whether it can do the job at all. Can it work with your computer. Can it work with your other tools. Can it transform your data into answers. A language should be complete on the first two, and minimize the unavoidable effort required for the last.

HS/FORTH opens your computer like no other language. You can execute function calls, DOS commands, other programs interactively, from definitions, or even from files being loaded. DOS and BIOS function calls are well documented HS/FORTH words, we don't settle for giving you an INT CALL and saying "have at it". We also include both fatal and informative DOS error handlers, installed by executing FATAL or INFORM.

HS/FORTH supports character or block, sequential or random i/o. The character stream can be received from 'sent to console, file, memory, printer or comfort. We include a communications plus upload and download utility, and foreground/background music. Display output through BIOS for compatibility or memory mapped for speed.

Our formatting and parsing words are without equal. Integer, double, quad, financial, scaled, time, date, floating or exponential, all our output words have string mapped for speed. Our formatting and parsing words are without equal. Integer, double, quad, financial, scaled, time, date, floating or exponential, all our output words have string mapped for speed.

HS/FORTH supports text/graphic windows for MONO thru VGA. Graphic drawings (line rectangle ellipse) can be absolute or scaled to current window size and clipped, and work with our perpqty routines. While great for plotting and line drawing, it doesn't approach the capabilities of Metawindows (tm Metagraphics). We use our Rosetta Stone Dynamic Linker to interface to Meta-windows.

HS/FORTH with MetaWindows makes an unbeatable graphics system. Or Rosetta to your own preferred graphics drivers.

HS/FORTH provides hardware/software floating point, including trig and transcendental. Hardware fp covers full range trig, log, exponential functions plus complex and hyperbolic counterparts, and all stack and comparison ops. HS/FORTH supports all 8087 data types and works in RADIONS or DEGREES mode. No coprocessor required. No problem. Operators (mostly fast machine code) and parse format words cover through 18 digits. Software fp eliminates conversion round off error and minimizes conversion time.

Single element through 4D arrays for all data types including complex use multiple cfa's to improve both performance and compactness. Z = (X,Y) (X + Y) would be coded: X Y .X Y + /IS Z (16 bytes) instead of: X Y Y Y .X @ Y .X Y + /I Z (26 bytes). Arrays can ignore 64k boundaries. Words use SYNONYM's for data type independence. HS/FORTH can even prompt the user for error on erroneous numeric input.

The HS/FORTH machine coded string library with up to 3D arrays is without equal. Segment spanning dynamic string support includes insert, delete, add, find, replace, exchange, save and restore string storage.

Our minimal overhead round robin and time slice multitaskers require a word that exits cleanly at the end of subtask execution. The cooperative round robin multi-tasker provides individual user stack segments as well as user tables. Controls pass to the next task/user whenever desired.

APPLICATION CREATION TECHNIQUES

HS/FORTH assembles to any segment to create stand alone programs of any size. The optimizer can use HS/FORTH as a macro library, or complex macros can be built as colon words. Full forward and reverse labeled branches and calls complement structured flow control. Complete syntax checking protects you. Assembler programming has never been so easy.

The Metacompiler produces threaded systems from a few hundred bytes, or Forth kernels from 2k bytes. With it, you can create any threading scheme or segmentation architecture to run on disk or ROM.

You can turnkey or seal HS/FORTH for distribution, with no royalties for turnkeyed systems. Or convert for ROM in saved, sealed or turnkeyed form.

HS/FORTH includes three editors, or you can quickly shave to your favorite program editor. The resident full window editor lets you reuse former command lines and save to or restore from a file. It is both an indispensable development aid and a great user interface. The macro editor provides reusable functions, cut, paste, file merge and extract, session log, and RECOMPILE. Our full screen Forth editor edits file or sector mapped blocks.

Debug tools include memory / stack dump, memory map, decompile, single step trace, and prompt options. Trace scope can be limited by depth or address.

HS/FORTH lacks a "modular" compilation environment. One motivation toward modular compilation is that, with conventional compilers, recompiling an entire application to change one subroutine is unbearably slow. HS/FORTH compiles at 20,000 lines per minute, faster than many languages link — let alone compile! The second motivation is linking to other languages. HS/FORTH links to foreign subroutines dynamically. HS/FORTH doesn't need the extra layer of files, or the programs needed to manage them. With HS/FORTH you have source code and the executable file. Period.

Development environments" are cute, and necessary for unnecessarily complicated languages. Simplicity is so much better.

HS/FORTH Programming Systems

Lower levels include all functions not named at a higher level. Some functions available separately:

- Documentation & Working Demo (3 books, 1000 + pages, 6 lbs) $ 95.
- Student $145.
- Personal optimizer, scaled & quad integer $245.
- Professional 80x87, assembler, turnkey, $395.
- RSDL linker, dynamic strings, multitasker $495.
- Physical screens $495.
- Production ROM, Metaeditor, Metawindows $949.
- Level upgrade, price difference plus $ 25.
- OBJ modules $495.
- Rosetta Stone Dynamic Linker $95.
- Metawindows by Metagraphics (includes RSDL) $145.
- Hardware Floating Point & Complex $ 95.
- Quad integer, software floating point $ 45.
- Time slice and round robin multitaskers $ 75.
- GigaForth (80286/386 Native mode extension) $295.

HARVARD

SOFTWORKS

PO BOX 69
SPRINGBORO, OH 45066
(513) 748-0390
When writing code for controllers, there are times when I need to enter mixed-precision parameters. Mixing double- and single-precision numbers gets messy, and if I forget to enter the period on a double number, there won’t be enough elements on the stack and the command will misbehave.

It would be nice to be able to enter numbers without thinking too much about periods and stack effects. If I enter a value that is 16 bits or less without a period, my code should be smart enough to put a zero in the high word value of a double-precision number. For instance, if a controller moves graphics from some large memory space to one of three video “pages,” I would like to be able to enter the command as:

<addr> <page> SHOW-VID

The usual way to handle this is to require <addr> to be a double-precision number:

100. 1 SHOW-VID
1A7BF000. 1 SHOW-VID

The more commands I have, the more difficult it is to keep track of the parameters which need periods. This process is especially error-prone when I am entering a value that can be represented by a single-precision number for a parameter that can handle double precision. It’s not so hard to remember to put the period after 1A7BF000, but when I enter 100, the period is easy to forget.

The Forth-79-defined INTERPRET has a very easy solution. See Figure One — note the DROP before [COMPIL] LITERAL. NUMBER always returns a double-precision number, and DPL is checked to see if a period was entered with the number. If a period wasn’t entered, the high 16 bits of the double number are dropped. If I enter 12345, then 2345 is left on the stack and the 1 is dropped. If I enter 1234, then a 0 is dropped.

“Words that use 16-bit values will see normal stack effects.”

But if that value could be saved and retrieved easily, I could always use a 16- or 32-bit number for any parameter without entering a period, and the Forth word that uses it could get the high 16 bits only if it needed them.

My solution is to construct a “shadow stack” to save the high 16 bits of any number entered (converted from ASCII by NUMBER, via INTERPRET). The shadow stack pointer always tracks the main parameter-stack pointer, so the low 16 bits and upper 16 bits of any number entered are always at the same relative position on the two stacks. The current value of the stack pointer also determines the stack pointer position for the shadow stack. As a consequence, any Forth word which leaves a number on the stack will also push the shadow stack pointer down, but the high 16 bits on the shadow stack will never be used — only numbers entered from the keyboard will produce usable entries on the shadow stack.

Now, any time a number is on the stack, I can retrieve the high word “shadow” by entering @SHADOW (as long as it was gotten via INTERPRET). I can also define some words to get the shadow word of the second and third numbers on the stack, called OVER@SHADOW and 3PICK@SHADOW.

Now, SHOW-VID can use the parameter stack and shadow stack — see Figure Two.

The only problem with this technique is that it only works while interpreting. I can’t simply compile a word like:

: CUE_FRAME
1775F000 3 SHOW-VID ;

This is because the 1775F000 would only be compiled as F000. The 1755 upper word would be lost, and the @SHADOW in SHOW-VID wouldn’t get the correct value.

This type of compilation is rare in my application, but sometimes it is required. So I extend the shadow stack when used in compilation with a new type of “shadow literal.”

The definition above would have to be compiled as:

: MY_WORD
[ 1775F000 ]S 3 SHOW-VID ;

I admit that this is not as compact as:

: MY_WORD
1775F000 . 3 SHOW-VID ;
But, since I am usually executing the word from an interpretive mode, the tradeoff makes sense. There are probably lots of extensions to this technique. For instance, a shadow stack could keep track of the position of the decimal point in a list of double-precision numbers.

This code could be used to make a pseudo-32-bit Forth. Any words that need to pass double-precision numbers can use the shadow stack, thereby eliminating the stack problems of mixed operators. The operation of +, -, *, and / could be redefined to deal with both the parameter stack and the shadow stack. This type of implementation would handle double-precision arithmetic in a different way. Words that only use the 16-bit values will see the normal stack effects, and words that need 32-bit results can get the high word off the shadow stack.

(Screens begin on next page.)

Darrel Johansen is currently a programmer engineer at Orion Intruments.

Figure One. Forth-79's INTERPRET per All About Forth (Glen B. Haydon, Mountain View Press).

```
: <INTERPRET>
  BEGIN -FIND
  IF STATE @ <
    IF CFA ,
    ELSE CFA EXECUTE
  THEN
  ELSE HERE NUMBER DPL @ 1+ 
    IF [COMPILE] DLITERAL
    ELSE DROP [COMPILE] LITERAL
  THEN
  \High bits are dropped
  THEN
  \ if no "." is entered.
  THEN
  AGAIN ;

  : INTERPRET <INTERPRET> ;
  \Allows redefinition
```

WE'RE LOOKING FOR A FEW GOOD HEADS.

DASH, FIND ASSOCIATES
Forth Recruiters
Under New Management

70 Elmwood Ave./Rochester, NY 14611/(716) 235-0168

COMPLETE DEVELOPMENT SYSTEM FOR MACHINE CONTROL APPLICATIONS

TINY188 is a low cost "PC somewhat compatible" engine for OEM controller applications. A selection of high level languages is available in ROM.

DDS188 An optional development board with EPROM programmer, floppy disk controller and added memory, removes to lower target system cost.

Prices start at $269 each/$99 at 1,000.

Vesta Technology, Inc.
(303) 422-8088
Figure Two. Example application of the "shadow stack."

0 \ Shadow stack for 32-bit number precision  drj 08Aug88
1 FORTH DEFINITIONS HEX
2 CREATE SHADOW_STACK 70 ALLOT \ Create a buffer area
3 \ for shadow stack.
4 SHADOW_STACK 6C + CONSTANT TOP_SHADOW \ Top of shadow stack.
5 6 : SHADOW_PTR TOP_SHADOW ( - n ) \ Compute shadow stack
6 \ pointer.
7 80 SP@ - - ; \ Leave addr on stack.
8 8 : !SHADOW_SHADOW_PTR ! ; ( n - ) \ Store into shadow stack.
9 A This screen creates a "shadow" stack that holds the upper
B 16 bits of a number when the number put on the stack is not
C entered with a "." to create a double number.
D This stack "tracks" the parameter stack with another pointer, so
E at any time, the high 16 bits of any number the user enters on
F the stack can be retrieved for any number on the stack.

0 \ replacement for <INTERPRET>  drj 08Aug88
1 : <SHADOW_INTERPRET>
2 BEGIN -FIND
3 IF STATE @ <
4 IF CFA ,
5 ELSE CFA EXECUTE
6 THEN
7 ELSE HERE NUMBER DPL @ 1+
8 IF [COMPILE] LITERAL
9 ELSE !SHADOW [COMPILE] LITERAL \ The only change
A THEN
B THEN
C AGAIN ;
D E <SHADOW_INTERPRET> CFA \ INTERPRET !
F

NOW FOR IBM PC, XT, AT, PS2
AND TRS-80 MODELS 1, 3, 4, 4P
The Gifted Computer

1. Buy MMSFORTH before year's end,
to let your computer work harder and
teacher.
2. Then MMS will reward it (and you)
with the MMSFORTH GAMES DISK.
a $39.95 value which we'll add on at
no additional charge!
MMSFORTH is the unusually smooth
and complete Forth system with the
great support. Many programmers report
four to ten times greater productivity
with this outstanding system, and MMS
provides advanced applications pro-
grams in Forth for use by beginners and
for custom modifications. Unlike many
Forths on the market, MMSFORTH gives
you a rich set of the instructions, editing
and debugging tools that professional
programmers want. The licensed user
gets continuing, free phone tips and a
MMSFORTH Newsletter is available.
The MMSFORTH GAMES DISK includes
arcade games (BREAKFORTH, CRASH-
FORTH and, for TRS-80, FREEWAY),
board games (OTHELLO and TIC-TAC-
FORTH), and a top-notch CRYPTO-
QUOTE HELPER with a data file of
coded messages and the ability to en-
code your own. All of these come with
Forth source code, for a valuable and
enjoyable demonstration of Forth pro-
gramming techniques.
Hurry, and the GAMES DISK will be our
free gift to you. Our brochure is free,
too, and our knowledgeable staff is
ready to answer your questions. Write,
Better yet, call 617/653-6136.

MMSFORTH and a free gift!

GREAT FORTH:
MMSFORTH V2.4 ................ $179.95*
The one you've read about in FORTH: A
TEXT & REFERENCE. Available for IBM
PC/XT/AT/PS2 etc., and TRS-80 M.1, 3
and 4
GREAT MMSFORTH OPTIONS:
FORTHWRITE ................... $99.95*
EXPERT-2 ................... 69.95
UTILITIES ................... 49.95
* Single-computer, single-user prices; corpo-
rate site licenses from $1,000 additional.
A $15 format, add $5/disk; Tandy 1000, add
$20. Add 5/6, plus 5% tax on Mass. orders.
DH+ not available for TRS-80s.
GREAT FORTH SUPPORT:
Free user tips, MMSFORTH Newsletter,
consulting on hardware selection, staff
training, and programming assignments
large or small.
GREAT FORTH BOOKS:
FORTH: A TEXT & REF. ........ $21.95*
THINKING FORTH .......... 16.95
Many others in stock.

MILLER MICROCOMPUTER SERVICES
61 Lake Shore Road, Natick, MA 01760
(617/653-6136, 9 am - 9 pm)
0 \ shadow stack, cont.   drj 08Aug88
1 : @SHADOW SHADOW_PTR 2- @ ; ( -n) \ Get top number on shadow stack.
2
3 : OVER @SHADOW SHADOW_PTR @ ; \ Get second number on shadow stack.
4 ( -n)
5 : 3PICK @SHADOW SHADOW_PTR 2+ @ ; \ Get third number on shadow stack.
6 ( -n)
7 exit
8
9 The SHADOW_STACK is simply a buffer area that can exist anywhere in memory. Under normal situations, it is never more than about 24 bytes deep. I have allotted it with eight more bytes than my parameter stack to give it a buffer for overflow and underflow, but !SHADOW should be modified to check boundaries making it completely bulletproof. This version is written for readability, but it works for almost any situation.

(Continued from page 5.)

to point to the first unused cell, and then it stores the number from the top of the parameter stack there. And the code for X> first fetches the top item from the extra stack to the parameter stack, and then decrements the stack pointer by two so that it points to the new top of the extra stack.

Several of the words here are simplified by the choice of having the stack pointer point to the cell: XCLEAR simply stores into this cell the cell’s own address, so it points to itself. Also, X$ does not have to adjust the pointer, but simply executes XSTACK @ @. XPICK and XDEPTH are also made simpler by this choice.

Screen seven redefines some of the stack words to include tests that all stack operations take place within the proper limits of the XSTACK array. These tests use the word X?, which gives an error comment if the top number on the parameter stack is not larger than the one underneath.

This screen can be loaded on top of the first one while debugging the program that uses the extra stack. The compiler comments during loading will remind the user to comment out the LOAD instruction for this screen on the load screen when the debugging is finished.
the tenth annual

FORML CONFERENCE

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

Following Thanksgiving, November 25–27, 1988

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

Theme: Forth and Artificial Intelligence

Artificial intelligence applications are currently showing great promise when developers focus on easy-to-use software that doesn't require specialized expensive computers. Forth's design allows programmers to modify the Forth language to support the unique needs of artificial intelligence. Papers are invited that address relevant issues such as:

- Programming tools for AI
- Multiusers and multitasking
- Management of large memory spaces
- Meeting customer needs with Forth AI programs
- Windowing, menu driven or command line systems
- Captive Forth systems—operating under an OS
- Interfacing with other languages
- Transportability of AI programs
- Forth in hardware for AI
- System security

Papers about other Forth topics are also welcome. Mail your abstract(s) of 100 words or less to FORML Conference, Forth Interest Group, P.O. Box 8231, San Jose, CA 95155. Completed papers are due November 1, 1988.

Asilomar is a wonderful place for a conference. It combines comfortable meeting and living accommodations with secluded forests on a Pacific Ocean beach. Registration includes deluxe rooms, all meals, and nightly wine and cheese parties.

RESERVATIONS FOR TENTH FORML CONFERENCE

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

Conference attendee in double room - $275 • Non-conference guest in same room - $150 • Children under 17 in same room - $100 • Infants under 2 years old in same room - free • Attendee in single room - $325

Register by calling the Forth Interest Group business office at (408) 277-0668 or writing to: FORML Conference, Forth Interest Group, P.O. Box 8231, San Jose, CA 95155.
Forth is based on the philosophy of keeping things simple. Remember that something elegant can be simple. Simplicity does not necessarily mean primitive. But vendors feel obligated to provide all of the utilities to which users have become accustomed. These are mutually exclusive goals. It is a dilemma for the Forth community to decide which goal it wishes to serve. It has been a dilemma for WISC Technologies, Inc. to decide how best to support its new computer architecture.

Keep it Simple

As many of you know, the origins of Forth were in running a real-time application at the "rug factory." There, the problem was the overhead imposed by the existing operating system. What evolved was a very simple, minimal program to avoid the overhead. In general, the simpler the program, the easier it is to maintain and support. Often it can be said that if 64Kb of program space is not enough, you probably do not understand the problem. Certainly, it is not efficient to allow programs to expand to fill the available space.

The Problem

The lack of acceptance of the Forth language among other computer programmers is perhaps due to their perceived requirement of having available all the tools they have come to expect in other programming languages. Though the WISC architecture was arrived at through an understanding of Forth, the writeable instruction set makes it a powerful generic processor. Reluctance to adopt it because of its association with Forth needs to be addressed, and a development environment designed to meet the expectations of programmers will be a big help.

WISC's design of the hardware has been strictly with the goal of keeping the architecture as elegantly simple as possible. As much as possible, assembling of hardware components with writeable instructions has been left to the programmer. Thus, the concept of a soft-wired system is used.

With the development of our products, the design has become less simple. For example, as we have progressed from the CPU/16 to the CPU/32, we have, of necessity, increased the complexity of the design. It is much easier to route a single bus than multiple buses, especially when each bus is 32 bits wide. As we have progressed to the engineering implementation in silicon, other capabilities have been added. With each addition, the balance between increased complexity and maintenance of simplicity has been considered.

Teaching Something New

When it comes to making the system available to others, demands are made for something other than simplicity. Since microcode must be written using some sort of development system, that system must meet the demands of those who will use it. As with the classical perspective of any teaching program, one must always start building on the knowledge and expectations the students already have.

In this case, the student is presumably a programmer already familiar with a computer system. He expects an operating system. He expects an editor or word processor of some sort. He expects a large library of built-in functions. He expects an effective debugger. After all, how could he write a program which did not need debugging? As illustrated by other papers at the Conference, some programmers know how to type while others would rather chase icons around the screen with a mouse or perhaps some combination of typing and a mouse. Any development system meeting all the above requirements may never meet the philosophical goals of Forth.

Programmer Psychology

In many ways, the computer operating system is a sort of language, along with the programming language and the natural language used by the programmer. Problems develop when the programmer is not a master of his system. Clearly, a person who is not a good touch typist will not be able to make effective use of keyboard commands. A person who does not have good dexterity with a mouse will find it a hindrance.

These limitations, in large part, depend upon the unique way each individual's nerves are connected, as well as his training. No two individuals are the same. I expect that part of the preference for a keyboard or a mouse is firmly rooted in the individual's capabilities and past experience. It will never be possible to make all individuals the same, but the psychological mind set of our potential users must be addressed.

Real World Requirements

In the real world, a development system is required that will attract many prospective users. It is certainly not necessary to call such a development system Forth. The first requirement is that the development system run on some existing hardware with an already defined operating system. WISC has started with a host
based on the Intel processor, in the form of an IBM PC, XT, or AT running under PC-DOS. The original implementation of Forth on these processors served as a basis. With the PC-DOS operating system comes a file structure for the storage devices, which in its latest versions include directories and subdirectories. Superimposed upon the initial Forth, then, is the requirement of an effective implementation of access to that system of files.

System files under PC-DOS are of variable length. Most programmers are already familiar with some sort of editor, often some variant of the well-known WordStar. But Forth, in its original form, had a rudimentary line editor and no files. The external storage device was directly accessed in sequential, fixed-length blocks of 1K bytes. No file structure was imposed. This conflicts with current operating systems. One answer is to access 1K blocks within system files, although that introduces more complexity. But using Forth blocks does have some advantages in the thinking process of the programmer. It tends to encourage factoring of problems.

However, this is more a consideration of programming style than the need of 1K blocks. In open, free-form text files, there is no imposed discipline, i.e., no style is imposed. What is needed is the development of a free-form style of programming in standard text files. With variable-length text files, it is no longer necessary to separate the source code from help screens or shadow screens. In addition, a set of text vectors could be added with each new function. All related material could be grouped in the same place. Thus, simply by adopting a style which includes all of the narrative documentation, source code, and text vectors as a unit, it will be more likely that the programmer will see that the documentation is maintained. It is a simple matter to compile the source code directly from such files and skip the documentation data. This approach is completely different from the experience of many Forth programmers. Other programmers are accustomed to a style of programming for each language they use. A special style should not come as a shock to them.

Standard Libraries

Forth has no standard library of extended functions. In part, this is because there has been no driving force as powerful as AT&T or IBM to produce the necessary libraries. But it must not be ignored that other programmers have come to expect these libraries.

An important consideration here is how these libraries are used. With the ease and speed of compilation in Forth, there is little need to save object code. Source code in the above-suggested style will provide all of the necessary documentation and can simply be compiled as needed. This avoids the necessity of a loader to make the object code relocatable.

What tools?

Forth as a new language is intimidating to many with previous programming experience. The educational system in this country has already indoctrinated students with BASIC, C, Pascal, FORTRAN, etc. in their many formal courses. The educational system does not offer students Forth as an alternative language. But Forth is no more complicated than BASIC or C as a programming language. It is just different, and people don’t like to change.

For a marketable development system, we must consider the potential user. If we want to survive, we must cater to his perceived needs. We may be confident he will not use many of the tools we might provide him after he learns to use the development system. But to sell the product, we must provide the perceived needs. For example, after the user has worked with the development system, he might learn that by utilizing the convenient factoring techniques of Forth and a set of appropriate test vectors, he will have little if any need of a debugging suite. But to start with, that possibility is beyond any experienced programmer’s conception. The problem is, who is going to take the time to write a program for sales purposes which he is confident will never be used?

Learning from Others

In spite of the opinions of many Forth programmers to the effect that non-Forth programmers have much to learn, it might just be that some Forth programmers could learn from the experience of others. This does not mean that a Forth programmer should make his Forth look like one of the other programming languages. The requirements imposed by compilers for other languages need not be imposed upon programs written in Forth.

Rather, the Forth programmer might discover new concepts if he were to understand why other languages are done the way they are. Most high-level languages were designed for specific applications with specific hardware. Some of these languages have migrated to other hardware and used for other applications. The more general the capabilities of a language compiler, the more complicated it becomes. Forth has the capability of incrementally adding new compiler instructions to the compiler along with the program. This is a new experience for many non-Forth programmers.

Actually, the C kernel is nearly as small as most Forth kernels. The C language must make use of a large number of add-on functions. In fact, the C kernel is so small that it cannot stand alone. There must at least be a set of standard I/O functions. There are more similarities than might at first be apparent.

Current WISC Development System

WISC is aggressively working on a development system for use with our hardware products. Many of these thoughts are addressed in the pieces we have already incorporated. Especially for the CPU/32, we must dispense with the 16-bit address space. A full, 32-bit, linear address space seems mandatory. Systems with only 16 bits of address space will soon be obsolete. Let the implementation map the 32-bit address space into the segment and offset requirements of the Intel processors, if that is what is being used.

Among the necessary libraries are a complete set of extended math and floating-point functions. Phil Koopman has written such a math library and placed it in the public domain. It includes efficient factoring and appropriate polynomials for the transcendental function. His library has been tested for over five years in a variety of applications, and appears to be essentially free of bugs. The floating-point values are based on the IEEE short form, already an accepted standard in computing circles. Large parts of the package have been published in Forth Dimensions. Some vendors have adopted it, but others are reinventing the wheel. In the interest of furthering a well-tested, standard set of extended math and floating-point functions, WISC is making copies of the source code on PC-DOS formatted disks available.
at this Conference. We hope this will contribute to the library of functions being demanded by those who might use Forth.

Another important set of functions should include an editor integrated with the compiler. Though it should be possible to use any editor or word processor to create the source files, having an integrated editor within the development system has many advantages. Such an editor could be used to indicate the beginning of an incremental compiling step, much like the use of the old blocks. And when compilation fails, the point of failure should be shown from within the editor, much like the WHERE function in older Forth implementations. As suggested above, an editor based on the common WordStar function might reduce the shock of having to learn another editor. WISC has such an editor operational, but it is not yet ready for release.

Other libraries might include a familiar-appearing debugger, a common set of string-handling utilities, and more. The structure of the development system should allow easy expansion with other libraries as they are identified, implemented, and well tested.

Conclusions
Keeping things simple does not mean that things are primitive. For the WISC processors to be accepted in the marketplace, we must provide an acceptable development system, even if that system appears quite different from conventional Forth. We should build on the experience we have had with Forth. WISC is well along in writing a development system along the lines described in this paper. There is really no dilemma. The system will be simply elegant.
While working on a project involving very extensive string manipulation, I realized that traditional Forth techniques for working with strings are impractical when anything more than nominal string manipulation is required. This paper describes words for the manipulation of strings using a string stack, including extensive pattern-matching support.

Over the last two years, I have worked extensively with the MUMPS computer language, for a good portion of that time implementing a non-standard, standalone MUMPS environment (interpreter/compiler/operating system) running in a multiprocessor Data General MV20000 environment; and writing systems tools in a standard MUMPS environment in a clustered DEC VAX environment.

Although MUMPS is a real dog in the performance department, it has many very useful features. The most striking characteristic of MUMPS is that all data types are treated as strings by the application programmer (although they are maintained as separate data types internally).

The project where I needed a string package was being done for Laboratory Microsystems, Inc.’s UR/Forth, PC/Forth 3.2, and PC/Forth+ 3.2. LMI’s versions of Forth have many excellent features, some of which are string related. However, the primary string-storage mechanism in the LMI products consisted of using a circular string buffer for temporary storage of strings. The problem with the string buffer approach is that the storage is very temporary. Since strings are stored in a circular fashion and the string buffer is used by the operating system interface, it is likely that extensive use of the string buffer will cause

```
CR . ( Loading STRING.4TH ) CR

( STRING.4TH rdb 10/24/87 )
( Last revised: rdb 11/30/87 )
( This file contains various string stack operators and assumes
the existence of several words not in the Forth-83 Standard: )
( -ROT, NIP, TUCK, NUMBER?, ?DO, CASE, OF, ENDOF, ENDCASE, >PTR)
( ADDR>PTR, C@L, C! L, CMOVEL, and CMOVEL>. )

( MAX$ -- addr
( Contains the maximum number of strings on the string stack. )

VARIABLE MAX$

( $0 -- addr
( Contains a pointer to the base of the string stack. )

VARIABLE $0

( $P -- addr
( Contains a pointer to the top of the string stack. )

VARIABLE $P

( $P! addr --
( Sets the address of the string stack pointer. )

: $P! ( addr -- )
  $P ! ;

( $INIT addr n -- addr'
( Initializes the string stack for n strings with addr as the
( highest address to use, returning the low address addr' used. )

: $INIT ( addr n -- addr' )
  TUCK MAX$ ! DUP $0 ! DUP $P! ( Stack base )
  SWAP 256 * - DUP MAX$ @ 256 * BLANK ; ( Clear stack )
```
an overflow situation, overwriting one
string with another.

Let me make clear, however, that the
string buffer approach is a very good one
and is useful for most applications. In this
case, though, I needed more permanent
temporary storage. What jumped to mind
was a string stack.

This idea did not spring at me out of the
clear blue sky. There have been at least
thirteen papers on the subject, according to
A Bibliography of Forth References (Third
Edition, The Institute for Applied Forth
Research). There have been numerous ar-
ticles in Dr. Dobb's Journal and Computer
Language, as well.

Most influential in this case was some
unpublished work by John James, whose
code provided a starting point for this pack-
age. The Forth community is frequently
accused of being more interested in rewrit-
ing Forth than in actually doing applica-
tions. I didn't want to fall into the trap of
reinventing the wheel one more time, but
matching beyond simple string compari-
sions. Therefore, I combined many of the
concepts I had encountered in MUMPS
into a Forth package that is much faster and
more compact than in a MUMPS environ-
ment.

What came out of all this was a series of
words which use a string stack and which
can be divided into the following groups:
memory words, stack words, manipulation
words, conversion and display words,
comparison words, defining words, and
miscellaneous words.

The only prerequisite for using these
words is to execute $INIT with the num-
ber of strings you want to allocate (I nor-
mally specify three) and the highest address
you wish to use — it will return the low
limit of the string stack area. I tend to
allocate work space starting from high
memory and work down, so this seemed
reasonable.

About the Source Code

The source code described in this docu-
ment assumes the existence of the follow-
ing words not in the Forth-83 Standard:
-RO T, NIP, TUCK, NUMBER?, ?DO, CASE, OF, ENDOF, ENDCASE, @DATE,
ADDR>PTR, >PTR, C@L, C@L, I!, CMOVE L, and CMOVE L. The definitions
of these words should be obvious and do
not need to be explained here, since they

```forth
( $OK? ...$ -- ...$ | )
( Verifies that the string stack has not under/overflowed. If )
( an error condition exists, an error message is displayed and )
( the string stack is reset. )

: $OK? ( ...$ -- ...$ | )
SP @ SP @ 2DUP U> -ROT ( Undeflow? )
MAX$ @ 256 * -- U< OR ( Overflow? )
IF $0 @ MAX$ @ TUCK 256 * -- SWAP $INIT ( Reinit )
CR "$ string stack under/overflow" ABORT ( Error )
THEN ;

( $SP$ $ -- $ str$^$ )
( Returns the address of the string on top of the string stack.)

: $SP$ ( $ -- $ str$^$ )
SP @ ;

( 1$ 1$ 0$ -- 1$ 0$ str$^$ )
( Returns the address of the string second on the string stack.)

: 1$ ( 1$ 0$ -- 1$ 0$ str$^$ )
SP @ COUNT + ;

( 2$ 2$ 1$ 0$ -- 2$ 1$ 0$ str$^$ )
( Returns the address of the string third on the string stack. )

: 2$ ( 2$ 1$ 0$ -- 2$ 1$ 0$ str$^$ )
SP @ COUNT + COUNT + ;

( N$ $.. n -- $.. str$^$ )
( Returns the address of the string nth on the string stack. )

: N$ ( $.. n -- $.. str$^$ )
SP @ SWAP 0 ?DO COUNT + SWAP 0 ( Get next str$^$)
LOOP ;

( $CNT $.. n -- $.. cnt )
( Returns the count cnt for string n on the string stack. )

: $CNT ( $.. n -- $.. cnt )
N$ C@ ;

( $DE PTH $.. -- $.. n )
( Returns the number n of strings on the string stack. )

: $DE PTH ( $.. -- $.. n )
SOK? 0 ( Over/under? )
BEGIN DUP N$ $0 @ U< ( Another? )
WHILE 1+ ( Inc cnt )
REPEAT ;
```
have been discussed at length in the literature. In addition, words manipulating date and time assume a specific format which is also explained in the code. This code runs on top of Laboratory Microsystems, Inc. UR/Forth, PC/Forth 3.2, and PC/Forth 3.2.

The basis for many of these words comes from John James' string package, written many years ago. The algorithm for $\text{SOUNDEX}$ came from Guy Kelly. The whole idea of $\text{SOUNDEX}$ dates back to the 1894 U.S. Census when they wanted to be able to find names that sounded alike. Since then it has been very widely used, but not adequately described. Although I could give the algorithm here, I think reading the code will do a better job of explaining it (I hope).

The source code for this package is in an ASCII file format, rather than in screens. I use this approach because of my philosophy about programming. Instead of concentrating on communicating with the machine, I try to communicate with programmers. The machine needs nothing more than a single space between each token; people need things like comments, indentation, phrasing, and so on.

In a standalone Forth environment, blocks are appropriate, since the goal is minimal overhead. In an environment using Forth running over an operating system, the operating system provides many services that we do not need to duplicate.

Most modern Forths have a word called $\text{INCLUDE}$ which loads the ASCII file named in the input stream. Two other words, $\text{SHELL}$ and $\text{~SHELL}$ provide hooks to the operating system, the first passing a string terminated by " to the command processor, while the second takes a counted string.

If you reserve a portion of RAM as a disk and copy a good editor to it on start up, the speed of calling the editor and editing short ASCII files is as fast in most cases as using a block editor. The editor I use is invoked with the word $\text{ED}$, which can be defined as in Figure One. Although there are some very nice block editors out there, I am still faced with the fact that I work on a variety of projects. Only having to use one editor in several different environments just makes my life easier.

Working with short ASCII files on a fast
system is little different from using block ranges in a traditional Forth environment and allows the use of specialized tools available under DOS, that I don’t have the time to recreate in Forth.

Another thing which may prove a little disconcerting to many people in the Forth community is what appear to be long definitions. Most of this comes from the fact that ASCII files allow a word which might be cramped on five or six lines to be spread out for readability. One of the definitions, $MATCH, really is quite long. In that case, the word is made up of a CASE statement within a $DO loop. I tried coding it several different ways, but this came out the cleanest. I hope the traditionalists will forgive me.

There is a rule of thumb used for decomposition: Don’t let a definition grow longer than one screen. I have a slightly different rule: Don’t let a definition grow longer than what you can easily hold in your head. If you have to write down the stack contents while you are coding or debugging, the word is too long. This means that some words which may be much less than one screen need to be further decomposed, while other words which are longer than one screen are just fine. The sole judge of this is you — just remember that your goal should be to communicate with another programmer, not just the machine.

With that in mind, here is the source code for this package. I hope it will prove useful to some of you. I have interjected some comments in spots in order to expand on the comments embedded in the code.

By the way, just because I have presented a very complete string stack word set, only the words actually used find their way into the final application. During development, I load the whole thing. When the application is complete, I comment out the words not used.

I hope this package is of interest to you. If you have suggestions or comments, I would really like to hear them. I can rewrite just about anybody’s code to make it tighter and faster. That goes for just about anybody else with my code. Together, we learn.

**Glossary of String-Stack Commands**

$! <addr> -- Stores the string on top of the string stack as a counted string at addr.

Code:

```
: $2! ( addr -- ^str )
   0 HERE 1+ ROT
BEGIN 2DUP C@ DUP 0<>
WHILE SWAP C! ROT 1+ ROT 1+ ROT 1+
REPEAT 4DROP HERE SWAP C! $@ ;
( Move to HERE )
( Until null )
( Inc cnt&ptr )
( Return $ )

( $! )
( Store the string $ on the string stack as an ASCII string )
( at addr, terminated by a null. )

( $2! )
( DUP $2! COUNT 2DUP + $! --ROT )
SWAP 2 PICK CMOVE + 0 SWAP C! ;
( Copy, $!0 )

( $INPUT -- $ )
( Accepts a character string of up to 255 characters from the )
( keyboard, creating a counted string $ on the string stack. )
( Input is terminated when either a Return character is found )
( or 255 characters have been input. )

( $INPUT ( -- $ )
   HERE DUP 255 SPAN @ $CNT@ ;

( $VARIABLE -- -- addr )
( Allocates memory for storage of a string. Used in the form: )
( $VARIABLE <name> )
( At compile time, $VARIABLE adds <name> to the dictionary and )
( ALLOTs memory for storage of a string in <name>’s parameter )
( field. When <name> is executed, it leaves its parameter field )
( address on the stack. The storage ALLOTed by $VARIABLE is not )
( initialized. )

( $VARIABLE ( -- ) ( -- addr )
   CREATE 256 ALLOT ;

( $CONSTANT str^ -- -- $ )
( Creates a string constant. Typically used in the form: )
( " <string>" $CONSTANT <name> )
( At compile time, $CONSTANT adds <name> to the dictionary and )
( compiles the counted string <string> in <name>’s parameter )
( field. When <name> is executed, <string> is left on the )
( string stack. )

( $CONSTANT CREATE ( str^ -- )
   DUP C@ 1+ TUCK HERE SWAP CMOVE ALLOT ( Save, allot )
   DOES> ( -- $ )
   $@ ; ( Get string )

( $NULL -- $ )
( Returns the null string <a zero length string> on the string )
( stack. )
```
: $NULL  ( -- $ )  
  SP@ 1- $P! 0 SP @ C! ;

($NULL?  $ -- $ flag
  Returns TRUE if the string on top of the string stack is the null string.
)  

: $NULL?  ($ -- $ f
  0 $CNT 0= ;

($DROP  $ --
  Discards the string on the top of the string stack.
)  

: $DROP  ($ -- )
  $P@ $CNT + $SP! ;

($2DROP  $ $ --
  Discards the top two strings on the string stack.
)  

: $2DROP  ($ $ -- )
  $SP$ SP@ COUNT + SP! ;

($DROP  $ -- $$
  ( Copies the string on top of the string stack to the top. )
)  

: $DROP  ($ -- $ $)
  $SP $@ ;

($2DUP  1$ SP$ -- 1$ SP$ SP$ 1$ SP$)
  ( Copies the top two strings on the string stack to the top. )

: $2DUP  ($ 1$ SP$ 1$ SP$)
  $SP$ DUP C@ 1+ 2DUP + C@ 1+ + 2DUP - DUP $SP! SWAP CMOVE ;

($OVER  1$ SP$ -- 1$ SP$ 1$)
  ( Copies the string second on the string stack to the top. )

: $OVER  ($ 1$ SP$ 1$ SP$ 1$)
  1$ $@ ;

($SWAP  1$ SP$ -- 0$ 1$)
  ( Exchange the top two strings on the string stack. )

: $SWAP  ($ 1$ SP$ 0$ 1$)
  1$ $@ 1$ SP$ 2DUP C@ SWAP C@ + 2+ CMOVE> $DROP ;

($PICK  $.. n -- $.. $n
  ( Copies the nth string on the string stack to the top. )
)  

$!L  $ ptr --
  Stores the string on top of the string stack as a counted string at the long address ptr.

$0  -- addr
  Returns the address of the variable which contains a pointer to the base of the string stack.

$2DROP  $ $ --
  Discards the top two strings on the string stack.

$2DUP  1$ SP$ -- 1$ SP$ 1$ SP$ 1$ SP$
  Copies the top two strings on the string stack.

$<$  1$ SP$ -- $ f
  Returns True if the second string (1$) on the string stack has a lower ASCII value than the first.

$=  1$ SP$ -- $ f
  Returns True if the top two strings on the string stack are equal.

$>  1$ SP$ -- $ f
  Returns True if the second string (1$) on the string stack has a greater ASCII value than the first.

$>D  $ -- $ d c
  Converts the string on the string stack to the double-precision integer $d$, using the current radix and the conversion count $c$.
  If all characters in the string are converted, $c$ is -1. If the string is partially converted, $c$ is the number of characters that converted.
  If $c$ is 0, the value of $d$ is undefined. The position of the decimal point is placed in the variable DPL. If no decimal point was present, DPL will contain the value -1. If either hardware or software floating-point extensions have been loaded, the action of $>D$ and the value in DPL may vary from this description.

$@  addr -- $  
  Fetches the counted string pointed to by address addr, returning the string to top of the string stack.

$@L  $ ptr -- $  
  Fetches the counted string pointed to by the long address (ptr), returning the string to top of the string stack.

$APPEND  1$ SP$ -- 2$  
  Appends the second string on the string stack (1$) to the string on top of the string stack.

$CMP  1$ SP$ -- $ f
  Compares the top two strings on the stack, returning a flag. If the second string (1$) is less than the first string, a -1 is returned. If the second string is greater than the first string, a 1 is returned.
Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Forth

Fortho
IF -1 ( $< )
ELSE 1 ( $> )
THEN ( )
ELSE 2DROP 0 ( $= )
THEN ( )
THEN 2DROP ;

( $= ) 1$ 0$ -- flag
(Returns TRUE if the top two strings on the string stack are equal.)

: $= ( 1$ 0$ -- flag )
$CMP 0 = ;

( $< ) 1$ 0$ -- flag
(Returns TRUE if the string second on the string stack has a lower ASCII value than the first.)

: $< ( 1$ 0$ -- flag )
$CMP -1 = ;

( $> ) 1$ 0$ -- flag
(Returns TRUE if the string second on the string stack has a greater ASCII value than the first.)

: $> ( 1$ 0$ -- flag )
$CMP 1 = ;

( $WITHIN ) 2$ 1$ 0$ -- flag
(Returns TRUE if the string 2$ on the string stack is greater or equal to string 1$ and less or equal to string 0$.)

: $WITHIN ( 2$ 1$ 0$ -- flag )
2 $SPICK $< NOT $< NOT AND ;

( $SUPPER ) $ -- $ ( Converts a string indicated by addr cnt to all upper case, not affecting any other ASCII symbols. )

: $SUPPER ( $ -- $ )
$SP@ COUNT OVER + SWAP ( limit start )
?DO I C@ ASCII ' OVER < ( char >= a )
OVER ASCII { < AND ( char <= z )
IF DUP 32 - I C! ( Convert )
THEN DROP ( Drop char )
LOOP ;

( $LOWER ) $ -- $ ( Converts a string indicated by addr cnt to all lower case, not affecting any other ASCII symbols. )

$INC 0$ -- 1$ ( Increments the lexicographical value of the string on the string stack, returning 1$.)

$INDEX 1$ 0$ -- o ( Returns the offset into the second string (1$) on the string stack of the first position matching the pattern of the first string. If 0$ is not a subset of 1$, -1 is returned. )

$INIT addr n -- addr' ( Initializes the string stack for n strings with addr as the highest address to use, returning the lowest address used. )

$INPUT -- $ ( Accepts a character string of up to 255 characters from the keyboard, creating a counted string on the string stack. Input is terminated when either a return character (0Dh) is found or 255 characters have been input. )

$JULIAN> $ -- y md ( Converts the Julian day of the string to the standard date format integers y md. The Julian day is the day offset from the start of the current year. The Julian date is the number of days since the last conjunction of the 28-year solar cycle and the 19-year lunar cycle, calculated to be January 1, 4713 B.C. On December 31, 1986, the Julian date was 2,446,796. )

$LEADING+ 0$ n chr -- 1$ ( Appends n leading characters (chr) to the string on the string stack, returning the string 1$. )

$LEADING- 0$ chr -- 1$ ( Discards chr leading characters from the string on the string stack, returning the string 1$. )

$LEFT 0$ n -- 1$ ( Extracts a string of length n from the string on the string stack, starting with the first character. The length of 1$ is the MIN of the length n with the length of 0$. )

$LOWER 0$ -- 1$ ( Converts the string on top of the string stack to all lower case, not affecting any other ASCII symbols, returning the modified string. )

$MATCH 1$ 0$ -- f ( Returns True if the string 1$ on the string stack matches the pattern of 0$. The pattern of 0$ may consist of the pattern codes of C, G, N, E, A, I, U, E, ~, or -. If the pattern code is a ~, the following character is taken as a literal value and True is returned if that character is present in the string 1$. If the pattern code is a ~, the following character is taken as a literal value and True is returned if that character is not present in the string 1$. The pattern is the union of the pattern codes in 0$. The significance of the pattern codes are: )
C 33 Control characters, including Del
G 128 Graphic characters above Del
N 10 Numeric characters
P 33 Punctuation characters, including SP
A 52 Alphabetic characters
L 26 Lower-case alphabetic characters
U 26 Upper-case alphabetic characters
E Everything non-graphic

'The following character is present
'- The following character is not present

$MID 0$ 1 -- 1$ Extracts a string of offset 0 and length 1 from the string on the string stack. If the offset 0 is greater than the length of string 0$, a null string is returned as 1$. The length of 1$ is the MIN of the length with the length of 0$ minus the offset.

$MM/DD/YY> $ y md
Converts the date string in the format mm/dd/yy to the standard date integers y md.

$SNIP 1$ 0$ -- 0$ Discards the second string on the string stack.

$NULL -- $
Returns the null string (a zero-length string) on the string stack.

$NULL? $ f
Returns True if the string on the string stack is null (a zero-length string).

$OK? --$
Verifies that the string stack has not under/overflowed. If an error condition exists, an error message is displayed and the string stack is reset.

$OVER 1$ 0$ -- 1$ 0$ 1$
Copies the second string on the string stack to the top.

$SP -- addr$
Returns the address of a variable which contains a pointer to the top of the string stack.

$SP! addr --$
Sets the string stack pointer to address addr.

$SP@ addr$
Returns the address of the current string stack pointer.

$PARSE 1$ 0$ -- 3$ 2$
Parses the string 1$ for the string 0$, returning the parsed string 2$ without the string 0$, and the remaining string 3$, also without the string 0$. If no instances of the string 0$ are found, string 2$ is the null string and string 3$ is 1$.

\begin{verbatim}
: $LOWER ( $ -- )
  $SP @ COUNT OVER + SWAP
  \end{verbatim}

\begin{verbatim}
: $APPEND ( 1$ 0$ -- 2$ )
  0 $CNT 1 $CNT +
  $SP @ COUNT OVER 1+ SWAP CMOVE>
  \end{verbatim}

\begin{verbatim}
: $LEFT ( 1$ 0$ -- 1$ )
  Extracts a string of length 1 from the string 0$ on the string stack, starting with the first character. The length of 1$ is the MIN of the length 1 with the length of 0$.
\end{verbatim}

\begin{verbatim}
$MID
0$ 1 -- 1$
Extracts a string of offset 0 and length 1 from the string on the string stack. If the offset 0 is greater than the length of string 0$, a null string is returned as 1$. The length of 1$ is the MIN of the length with the length of 0$ minus the offset.

$MM/DD/YY> $ y md
Converts the date string in the format mm/dd/yy to the standard date integers y md.

$SNIP 1$ 0$ -- 0$
 Discards the second string on the string stack.

$NULL --$
Returns the null string (a zero-length string) on the string stack.

$NULL? $ f
Returns True if the string on the string stack is null (a zero-length string).

$OK? --$
Verifies that the string stack has not under/overflowed. If an error condition exists, an error message is displayed and the string stack is reset.

$OVER 1$ 0$ -- 1$ 0$ 1$
Copies the second string on the string stack to the top.

$SP -- addr$
Returns the address of a variable which contains a pointer to the top of the string stack.

$SP! addr --$
Sets the string stack pointer to address addr.

$SP@ addr$
Returns the address of the current string stack pointer.

$PARSE 1$ 0$ -- 3$ 2$
Parses the string 1$ for the string 0$, returning the parsed string 2$ without the string 0$, and the remaining string 3$, also without the string 0$. If no instances of the string 0$ are found, string 2$ is the null string and string 3$ is 1$.

\begin{verbatim}
: $LOWER ( $ -- )
  $SP @ COUNT OVER + SWAP
  \end{verbatim}

\begin{verbatim}
: $APPEND ( 1$ 0$ -- 2$ )
  0 $CNT 1 $CNT +
  $SP @ COUNT OVER 1+ SWAP CMOVE>
  \end{verbatim}

\begin{verbatim}
: $LEFT ( 1$ 0$ -- 1$ )
  Extracts a string of length 1 from the string 0$ on the string stack, starting with the first character. The length of 1$ is the MIN of the length 1 with the length of 0$.
\end{verbatim}

\begin{verbatim}
$MID
0$ 1 -- 1$
Extracts a string of offset 0 and length 1 from the string on the string stack. If the offset 0 is greater than the length of string 0$, a null string is returned as 1$. The length of 1$ is the MIN of the length with the length of 0$ minus the offset.

$MM/DD/YY> $ y md
Converts the date string in the format mm/dd/yy to the standard date integers y md.

$SNIP 1$ 0$ -- 0$
Discards the second string on the string stack.

$NULL --$
Returns the null string (a zero-length string) on the string stack.

$NULL? $ f
Returns True if the string on the string stack is null (a zero-length string).

$OK? --$
Verifies that the string stack has not under/overflowed. If an error condition exists, an error message is displayed and the string stack is reset.

$OVER 1$ 0$ -- 1$ 0$ 1$
Copies the second string on the string stack to the top.

$SP -- addr$
Returns the address of a variable which contains a pointer to the top of the string stack.

$SP! addr --$
Sets the string stack pointer to address addr.

$SP@ addr$
Returns the address of the current string stack pointer.

$PARSE 1$ 0$ -- 3$ 2$
Parses the string 1$ for the string 0$, returning the parsed string 2$ without the string 0$, and the remaining string 3$, also without the string 0$. If no instances of the string 0$ are found, string 2$ is the null string and string 3$ is 1$.

\begin{verbatim}
: $LOWER ( $ -- )
  $SP @ COUNT OVER + SWAP
  \end{verbatim}

\begin{verbatim}
: $APPEND ( 1$ 0$ -- 2$ )
  0 $CNT 1 $CNT +
  $SP @ COUNT OVER 1+ SWAP CMOVE>
  \end{verbatim}

\begin{verbatim}
: $LEFT ( 1$ 0$ -- 1$ )
  Extracts a string of length 1 from the string 0$ on the string stack, starting with the first character. The length of 1$ is the MIN of the length 1 with the length of 0$.
\end{verbatim}

Subscribe to the Journal of Forth Application and Research like your colleagues and competitors!

To order Volume 5 of the Journal of Forth Application and Research call the Institute for Applied Forth Research, Inc. at:
(716) 235-0168

INDIVIDUAL: $50 CORPORATE: $125
Outside N. America: $70 Outside N. America: $140

Visit our booth at the FIG convention in Anaheim.
: SLEPT ( OS l -- l$ )
  0 $CNT OVER - DUP 0> ( #Drop > 0? )
IF $P@ 2DUP + 3 PICK 1+ CMOVE> ( Slide l chrs )
$P@ + DUP $P! $P C! ( Adj ptr, cnt )
ELSE 2DROP THEN ; ( Do nothing )

( SRIGHT OS l -- l$ )
( Extracts a string l$ of length l from the string OS on the )
( string stack, starting from the last character. The length of )
( l$ is the MIN of the length l with the length of OS. )

: SRIGHT ( OS l -- l$ )
  0 $CNT TUCK MIN TUCK - $P@ + $P! $P$ C! ;

( $MID OS o l -- l$ )
( Extracts a string l$ of offset o and length l from string OS$ )
( on the string stack. If the offset o is greater than the )
( length of the string SI, the null string is returned as l$. )
( The length of l$ is the MIN of the length l with the length )
( of OS minus the offset o. )
( $MID is equivalent to the LMI word STRXTR )

: MID ( OS o l -- l$ )
  0 $CNT ROT - DUP 0> ( o > len? )
IF SRIGHT 0 $CNT MIN SLEPT ( Extract str )
ELSE 2DROP $DROP $NULL THEN ; ( Return null )

( $CNT@L ptr cnt -- $ )
( Copies cnt characters of the string at the long address ptr )
( to the string stack, converting it to a counted string. )

: $CNT@L ( ptr cnt -- $ )
  >R $P@ R@ - DUP 1- $P! ( Adj stk ptr )
  ADDR>PTR R@ CMOVE! R> $P$ C! ; ( Move, !cnt )

( $CNT!L $ ptr cnt -- )
( Stores cnt characters of the string $ on top of the string )
( stack at the long address ptr. If cnt is greater than the )
( number of characters in $, the excess character positions are )
( blank filled. If cnt is less than the number of characters in )
( $, the string is truncated to cnt. )

: $CNT!L ( $ ptr cnt -- )
  0 $CNT OVER - 0> ( Pad w/sp? )
IF DUP PAD C! PAD 1+ SWAP BLANK ( Make sp str )
  PAD $@ $SWAP $APPEND ( Pad $ )
THEN $P@ COUNT OVER + $P! ( Drop $ )
  ADDR>PTR 2SWAP 0 $CNT CMOVE! ; ( Move string )

( S@L ptr -- $ )
( Fetches the string pointed to by the long address ptr to the )
( top of the string stack. )

$PICK $.. n -- $.. n$
Copies the nth string on the string stack to the top.

$RIGHT OS n -- l$
Extracts a string of length n from the string on the )
( string stack, starting from the last character. The length of )
( l$ is the MIN of the length n with the length of OS. )

$ROT 2S 1S OS -- 1S OS 2S$
Rotates the third string on the string stack to the top.

$SOUNDEX OS -- l$
Returns the soundex code string of the string on the )
( string stack. The soundex code consists of the first )
( character of the string, followed by a three-digit code in )
( the range 0000 ≥ l$ ≥ Z999. )

$SWAP 1S OS -- OS 1S$
Exchange the top two strings on the string stack.

$STRAILING+ OS n chr -- l$
Append n trailing characters chr to the string on the )
( string stack, returning the string l$. )

$STRAILING- OS chr -- l$
Discard n trailing characters chr from the string on the )
( string stack, returning the string l$. )

$STUCK 1S OS -- OS 1S OS$
Copies the string on the top of the string stack to the )
( third position on the string stack. )

$SUPPER $ -- $'
Converts the string on top of the string stack to )
( all upper case, not affecting any other )
( ASCII symbols, returning the modified string $'. )

$VARIABLE -- str$
Allocates memory for storage of a string.
Used in the form:
$VARIABLE <name>
and
$VARIABLE

At compile time, $VARIABLE adds <name> )
( to the dictionary and ALLOTS memory for )
( storage of a string in <name> 's parameter )
( field. When <name> is executed, it leaves its )
( parameter field address on the stack. The )
( storage ALLOTTed by $VARIABLE is not )
( initialized. )

$VERIFY 1S OS -- o$
Returns the offset into the second string (1S) )
( on the string stack of the first character in the )
( first string (OS) which is not found in the )
( second (i.e., the length of the initial substring of )
( OS which consists entirely of characters in 1S). )
$YMMDD>  $ -- y md
Converts the date string in the format
YMMDD to the standard date format
integers y md.

$Z:  $ addr --
Stores the string on the top of the string stack
as an ASCII string at addr, terminated by a
null.

$Z1L  $ ptr --
Stores the string on the top of the string stack
as an ASCII string at the long address ptr,
terminated by a null.

$Z8  addr -- $ mode
Returns the string on the string stack of the
ASCII string at the address addr which is
terminated by a null.

$Z8L  ptr -- $ mode
Returns the string on the string stack of the
ASCII string at the long address ptr which is
terminated by a null.

-S! 2$ 1$ 0$ -- 0$ 2$ 1$ mode
Rotates the top string on the string stack to the
third position.

$  $ -- $ mode
Displays the string on the top of the string
stack.

$.  $.. -- $.. mode
Non-destructively displays the contents of the
string stack.

1$ 1$ 0$ -- 1$ 0$ addr
Non-destructively returns the address of the
string second on the string stack.

Advertisers Index

Bryte - 37
Concept - 25
Dash-Find - 8
FIG Convention - 30
Forth Interest Group - 40
FORML - 11
Harvard Softworks - 6
Inner Access - 14
Inst. Forth Application and
Research - 22
Laboratory Microsystems - 33
Miller Microcomputer Services - 9
Next Generation Systems - 10
Silicon Composers - 2
Vesta Technology - 8

Forth Dimensions 24 Volume X, Number 3
: SDEC ( 0$ -- 1$ )
  SP@ COUNT ?DUP ( Check for null)
  IF 1- DUP C@ DUP 0 > ( > null? )
  IF 1- SWAP C! ( Dec char val )
  ELSE DROP 0 ?CNT DUP 1 >
  IF 1- $LEFT 255 SP@ COUNT + 1- C! ( Set to 255 )
  ELSE 2DROP $DROP $NULL THEN ( Return null )
THEN THEN ;

(Screen continued on page 37.)

Figure One.
Editor Word rdb 11/29/85

This word allows the use of an ASCII editor (or any other by
simply changing the name) from within UR/FORTH.

: ED ( -- )
" " SP $PARSE
" " ED $@ $CAT
$P@ ~SHELL $DROP ;

$ 1$ 0$ -- 2$ 1$ 0$ addr
Non-destructively returns the address of the
string third on the string stack.

>$YYYYMMDD y md --$
Converts the standard date format integers y
md to the date string, with the format
yyymmdd.

>D$ d --$
Converts the double-precision integer d to the
string on the string stack.

>MAX$ -- addr
Returns the address of variable which contains
the maximum number of strings on the string
stack.

>N$ n -- addr
Returns the address of the nth string on the
string stack.

F83 USERS

PVM83 is a complete Prolog extension to Laxen and Perry F83.
It handles the primary data structures of strings, numbers, logical constants,
logical variables, compound predicates, and lists. PVM83 is designed to add
productivity and flexibility. It is fully interactive between Prolog procedures,
and Forth code. PVM83 is a compiled Prolog featuring fast execution times.

PVM83 is fully extensible. "Standard" definitions give the
programmer flexibility to design just those procedures needed for his
application. PVM83 code can execute Forth words. F83 can call the PVM83
backtracking and problem solving capabilities.

PVM83 code is incrementally
compiled in higher memory segments
than the F83 core, leaving room in
the F83 kernel for the "standard" extensions
or other F83 code that the pro-
grammer needs.

PVM83 is designed to keep the
Forth philosophy of being both
compiled, and interactive. You can type
in procedures from the keyboard and
test them, or supply source code from
Forth block files, or text files
Intersegment memory management
source code included.

F83 USERS

PVM83 is a complete Prolog extension to Laxen and Perry F83.
It handles the primary data structures of strings, numbers, logical constants,
logical variables, compound predicates, and lists. PVM83 is designed to add
productivity and flexibility. It is fully interactive between Prolog procedures,
and Forth code. PVM83 is a compiled Prolog featuring fast execution times.

PVM83 is fully extensible. "Standard" definitions give the
programmer flexibility to design just those procedures needed for his
application. PVM83 code can execute Forth words. F83 can call the PVM83
backtracking and problem solving capabilities.

PVM83 code is incrementally
compiled in higher memory segments
than the F83 core, leaving room in
the F83 kernel for the "standard" extensions
or other F83 code that the pro-
grammer needs.

PVM83 is designed to keep the
Forth philosophy of being both
compiled, and interactive. You can type
in procedures from the keyboard and
test them, or supply source code from
Forth block files, or text files
Intersegment memory management
source code included.

PVM 83

only $69.95
includes manual
requires
DOS 2.0 or higher 266K RAM

Concept 4
PO Box 20136
VOC Az 86341

Volume X, Number 3
Forth Dimensions
SOME WORDS ABOUT F83's WORDS

TIMOTHY HUANG - PORTLAND, OREGON

[Editor's note: This was originally presented by its author at a FORML event in Taiwan (ROC). In some respects it shows the passage of time, but it does provide an avenue for the intermediate Forth programmer who is looking for a way to whet his skills on F83's kernel. Perhaps other uses of the technique will suggest themselves...]

This paper describes an alternate method of implementing F83's WORDS mechanism. The new implementation allows the original function of WORDS—which displays every word of a given vocabulary—and a display that is limited to important keywords.

Introduction
F83 is an excellent public-domain implementation of Forth-83. I love it and believe all who use it will agree with me. In my personal opinion, it is one of the best products available, for the following reasons:

- It is a very complete implementation of a software development environment, including many tools and utilities (e.g., metacompiler, multitasker) not provided in the basic packages of some commercial vendors.
- It not only complies with the Forth-83 Standard, it provides a unified image across several different microprocessors and operating systems. I.e., 6502 (Apple-DOS), 8086/88 (MS-DOS, CP/M-86), 8080/85 and Z80 (CP/M-80), and 68000 (CP/M-68K).
- It is widely available. You can download it from a computer bulletin board, copy it free from a friend, or pay $25 to No Visible Support Software.

However, there is also room for improvement. For example:

- Some utilities can be made more flexible (e.g., see this article).
- Several definitions should be rewritten in low level to improve performance (e.g., all stack operators should be in low level).
- Several custom utilities should be added for a given environment, to take advantage of built-in system features (e.g., the Escape sequence terminal control words should be used to provide a more friendly, screen-oriented editor—I hate the intolerable hostility of the line editor).

"Our lives (and our computers) are complicated enough."

- 32-bit implementations are needed for newer machines which incorporate the desktop metaphor (e.g., Apple's Macintosh, Atari's 520ST, Digital Research's GEM, and Commodore's Amiga).

A Better Way With WORDS
I like the name "WORDS." It demonstrates good taste and good Forth discipline. It is good English, easy to understand, powerful, and more human than FORTH's VLIST, which is not an English word but computer gibberish. This is a subtle but important human factor to consider.

Next, because F83 is quite a complete development system, it contains many, many words in several vocabularies. According to my unofficial statistics, there are more than 1000 words in the system. This is an overwhelmingly large number. In the FORTH vocabulary, there are more than 500 words. Somehow, my mind encounters great difficulty in remembering these names. Thus, a better mechanism must be found.

Finally, an unscientific analysis reveals that many of those words were created only for the ease and clarity they bring to the definition of other, more powerful and commonly used, words. In my situation, about 50% of the words could be hidden, reducing to a manageable level the number of words I need to be familiar with.

Selection Criteria
Whether any given word is better off hidden is up to personal taste, the requirements of a given task, and even the user's emotional state. Nevertheless, I would like to provide the following guidelines as general selection criteria.

Consider hiding words that fit any of the following categories:

- Words enclosed with parentheses, usually run-time routines or lower-level vectored words. These words won't be executed directly including, but not limited to, (WHERE), (CONVEY), (COPY), (PAUSE), (FORGET), (?DO), (DO), (LOOP), (+LOOP), and (LIT).
- Words used only in very limited circumstances. Dr. C.H. Ting's paper about the frequency of F83 word use (Forth Dimensions VII/4) is a good reference.
\ Load Screen
1 \ If you like my WORDS, use the following line.
2 3 4 THRU
3 5 LOAD
4
5 \ If you don't like my WORDS, use the latest from Mike Perry.
6 \ 2 LOAD
7
8
9
10
11
12
13
14
15

\ Display the WORDS in the Context Vocabulary
1 : LARGEST (S addr n -- addr ' val ) OVER \ SWAP ROT \ DO
2 2DROP @ U: IF -ROT 2DROP DUP @ OVER THEN 2* LOOP DROP ;
3
4 CREATE THREADS \THREADS \ ALLOT
5 : FOLLOW (S \ a:fl -- ) \THREADs \THREADS \ ALLOT
6 : another (S -- anf ) \THREADs \THREADS \ LARGEST DUP
7 IF DUP @ ROT! \NAME ELSE NIP THEN ;
8 DEFER EACH (S \ anf -- )
9 : EVERY (S \ avoc -- ) NEWLINE FOLLOW
10 : BEGIN START/STOP another \ DUP WHILE EACH \ REPEAT;
11
12 : \NAME (S \ anf -- ) DUP CB 31 AND \LINE 10 \ SPACES ;
13 : WORDS (S \ -- ) \NAME \ IS EACH \ CONTEXT \ EVERY ;
14 \ROOT \ DEFINITIONS
15 : WORDS \ WORDS ; \ FORTH \ DEFINITIONS

\ Display the WORDS in the Context Vocabulary - 1
1 : LARGEST (S addr n -- addr ' val ) OVER \ SWAP ROT \ DO
2 2DROP @ U: IF -ROT 2DROP DUP @ OVER THEN 2* LOOP DROP ;
3 CREATE THREADS \THREADS \ ALLOT
4 : FOLLOW (S \ a:fl -- ) \THREADs \THREADS \ ALLOT
5 : another (S -- anf ) \THREADs \THREADS \ LARGEST DUP
6 IF DUP @ ROT! \NAME ELSE NIP THEN ;
7 DEFER EACH (S \ anf -- )
8 : EVERY (S \ avoc -- ) NEWLINE FOLLOW
9 : BEGIN START/STOP another \ DUP WHILE EACH \ REPEAT;
10 : \NAME (S \ anf -- ) DUP CB 31 AND \LINE 10 \ SPACES ;
11 : Change and/or add the followings.
12 : ?TAG (S \ anf -- \ anf ! ) DUP CB 32 ANC ;
13 : SOME (S \ anf -- ) \TAG IF DROP ELSE \NAME THEN ;
14 : ALL (S \ -- ) \NAME \ IS EACH ;
15 : SOME (S \ -- ) \NAME \ IS EACH ;

\ Display the WORDS in the Context Vocabulary - 2
1 : LARGEST (S addr n -- addr ' val ) OVER \ SWAP ROT \ DO
2 2DROP @ U: IF -ROT 2DROP DUP @ OVER THEN 2* LOOP DROP ;
3 CREATE THREADS \THREADS \ ALLOT
4 : FOLLOW (S \ a:fl -- ) \THREADs \THREADS \ ALLOT
5 : another (S -- anf ) \THREADs \THREADS \ LARGEST DUP
6 IF DUP @ ROT! \NAME ELSE NIP THEN ;
7 DEFER EACH (S \ anf -- )
8 : EVERY (S \ avoc -- ) NEWLINE FOLLOW
9 : BEGIN START/STOP another \ DUP WHILE EACH \ REPEAT;
10 : \NAME (S \ anf -- ) DUP CB 31 AND \LINE 10 \ SPACES ;
11 : Change and/or add the followings.
12 : ?TAG (S \ anf -- \ anf ! ) DUP CB 32 ANC ;
13 : SOME (S \ anf -- ) \TAG IF DROP ELSE \NAME THEN ;
14 : ALL (S \ -- ) \NAME \ IS EACH ;
15 : SOME (S \ -- ) \NAME \ IS EACH ;

\ Display the WORDS in the Context Vocabulary - 3
1 : LARGEST (S addr n -- addr ' val ) OVER \ SWAP ROT \ DO
2 2DROP @ U: IF -ROT 2DROP DUP @ OVER THEN 2* LOOP DROP ;
3 CREATE THREADS \THREADS \ ALLOT
4 : FOLLOW (S \ a:fl -- ) \THREADs \THREADS \ ALLOT
5 : another (S -- anf ) \THREADs \THREADS \ LARGEST DUP
6 IF DUP @ ROT! \NAME ELSE NIP THEN ;
7 DEFER EACH (S \ anf -- )
8 : EVERY (S \ avoc -- ) NEWLINE FOLLOW
9 : BEGIN START/STOP another \ DUP WHILE EACH \ REPEAT;
10 : \NAME (S \ anf -- ) DUP CB 31 AND \LINE 10 \ SPACES ;
11 : Change and/or add the followings.
12 : ?TAG (S \ anf -- \ anf ! ) DUP CB 32 ANC ;
13 : SOME (S \ anf -- ) \TAG IF DROP ELSE \NAME THEN ;
14 : ALL (S \ -- ) \NAME \ IS EACH ;
15 : SOME (S \ -- ) \NAME \ IS EACH ;

\ Display the WORDS in the Context Vocabulary
1 : LARGEST (S addr n -- addr ' val ) OVER \ SWAP ROT \ DO
2 2DROP @ U: IF -ROT 2DROP DUP @ OVER THEN 2* LOOP DROP ;
3 CREATE THREADS \THREADS \ ALLOT
4 : FOLLOW (S \ a:fl -- ) \THREADs \THREADS \ ALLOT
5 : another (S -- anf ) \THREADs \THREADS \ LARGEST DUP
6 IF DUP @ ROT! \NAME ELSE NIP THEN ;
7 DEFER EACH (S \ anf -- )
8 : EVERY (S \ avoc -- ) NEWLINE FOLLOW
9 : BEGIN START/STOP another \ DUP WHILE EACH \ REPEAT;
10 : \NAME (S \ anf -- ) DUP CB 31 AND \LINE 10 \ SPACES ;
11 : Change and/or add the followings.
12 : ?TAG (S \ anf -- \ anf ! ) DUP CB 32 ANC ;
13 : SOME (S \ anf -- ) \TAG IF DROP ELSE \NAME THEN ;
14 : ALL (S \ -- ) \NAME \ IS EACH ;
15 : SOME (S \ -- ) \NAME \ IS EACH ;
Personally, I think words that are used fewer than 20 times in the system should be hidden, but exceptions exist. "MISSING" was used 113 times by other F83 words, but during several years I have never needed it. On the other hand, DUMP isn't used by any other words, but my programming life would be miserable without it. Personal judgment should be your guide.

- Non-English words should be hidden. Our lives (and our computers) are complicated enough without unnecessary burdens. Poor choices for word names just exposes a lazy programmer and will make your programs more difficult to maintain. I cannot see anything good about using non-English Forth word names.

### Syntax

Bearing these points in mind, we will now derive a method to provide a discriminatory word-listing mechanism. With this new way of dealing with words in the system, some words are tagged and some remain unchanged.

The first thing that requires our attention is the desired syntax. Because the main interpreter interfaces with us, it should be as close as possible to our human linguistic habits for ease of use and understanding. Awkward, unnatural, and non-English syntax will only degrade and prevent human progress. Thus, it is very important to get this right. At this point, I do not even care about all the other, nitty-gritty factors. What I care about is the overall approach to the whole thing.

I think the following two examples are great. They are short, simple, and plain English. Even we (Chinese) have no sweat comprehending them:

```
ALL FORTH WORDS
SOME FORTH WORDS
<choice> <vocabulary> <function>
```

The first phrase is equivalent to the old FORTH WORDS. It displays all words in the FORTH vocabulary, regardless of the nature of the words. The whole phrase only needs to be used once, to select the vocabulary and the behavior of WORDS; after that, just typing WORDS is enough. Of course, the vocabulary name — FORTH, in this example — can be that of any vocabulary.

The second phrase is the selective word listing. It only shows words that are not tagged. Again, after the complete
phrase has been typed once, WORDS will conform to that behavior until changed.

Glossary
The following words will be needed to implement this new mechanism:

**ALL (---)**
Vectors the function of WORDS so that all the words in a given vocabulary will be displayed.

**SOME (- -)**
Vectors the function of WORDS so that only the untagged words in a given vocabulary will be displayed.

**TAG (---)**
Marks a given word so that it will be hidden from the selective display.

Used in the form:

```plaintext
: <name> <definition> ; TAG
```

**SOME (- -)**
Displays only untagged words in the context vocabulary. Press any key to interrupt the listing.

```plaintext
?TAG ( nfa - - nfa flag )
```
Checks the count byte of the name field to see if the tag bit is set. If it is, returns true; otherwise, returns false.

**Implementation Notes**
In order to get the job done right, I think the following must be carefully considered during implementation:

- The normal dictionary search by the interpreter should not be affected by this implementation. I.e., to the interpreter, these two categories of words should maintain the same weight. This means that words like ` (tick), FIND, FORGET, etc. should be able to find all words in the dictionary, tagged and untagged alike.
- For the tag bit, it can use the smudge bit of the name field’s count byte. If this bit is set, then **SOME <voc>** WORDS should not see it. If it is not set, then **SOME <voc>** WORDS will see it. Regardless of the condition of the tag bit, **ALL <voc>** WORDS will display all the words in that vocabulary.
- Modification of the system should be kept to a minimum, if possible. As it turns out, except for the modifications in screen 5 of the UTILITY.SCR file, there is only one other place that must be changed.

**Order**

| Context: COUNTIES COUNTIES FORTH ONLY
| Current: FORTH

**All counties words**

TAINAN-COUNTY TAIPEI-COUNTY TAINAN_CITY MAR-DOU DAN-SHAY PEI-TOU SHAN-CHUNG

**Some counties words**

TAINAN-COUNTY TAIPEI-COUNTY

**Words**

TAINAN-COUNTY TAIPEI-COUNTY

**Taipei-county**

Shan-Chung is a city of very stinky river.
Pei-Tou is where the hot spring place.
Dan-Shay is an old harbor town with nice university.

**Tainan-county**

Mar-Do produces the best grape-fruit.
The old mayor is called Big-Head.

` Mar-Do . 22459

Figure One. Use of the selective WORDS, based on the example in screen five. Bold indicates keyboard entries.
Real-Time Programming Convention

November 18 - 19, 1988
Grand Hotel, Anaheim, California

Call for Presentations

The 1988 Real-Time Programming Convention will be held at the Grand Hotel in Anaheim, California, and is sponsored by the Forth Interest Group.

The theme of this year's convention is Real-time Programming Systems. The invited speakers are Jef Raskin, head of the original Macintosh development team and inventor of the Canon Cat, and Ray Duncan, well-known author and expert on IBM PC Operating Systems. Both speakers have made extensive use of Forth, a language especially suited to real-time applications.

There is a call for presentations on topics in the following areas:

Programming Environments
- Real-time Operating Systems
- Language-oriented RISC machines
- Parallel Processing
- Languages for Data Acquisition and Analysis
- Robotics and Real-time Device Control

Applications
- Aerospace
- Medical
- Laboratory
- Machine-vision
- Digital Signal Processing
- Robotics
- Automation
- Instrumentation

Intelligent Devices
- Intelligent Instrumentation
- Working Neural Nets
- Adaptive devices
- Software Peripheral Controllers

Presentations may be either talks or demonstrations. Talks are limited to fifteen minutes. Please submit an abstract of the talk and a request for any audio-visual assistance by October 15. Demonstrations may accompany the talk or appear separately throughout the convention. Please send a description of the demonstration and its requirements by October 15.

Abstracts and descriptions should be sent to: Real-Time Programming Convention, Forth Interest Group, PO Box 8231, San Jose, CA 95155.
Part Two

DESIGNING DATA STRUCTURES

MIKE ELOLA - SAN JOSE, CALIFORNIA

The Search for Portable Data Objects

Various design techniques can make the porting of data structures easier. Usually, the use of such design techniques reflects a general programming philosophy rather than a concern for portability.

The primary focus of this discussion is the portable coding of data objects. Because of their effect upon portability, certain programming techniques will become the focus of the following discussion. These techniques are: object-oriented programming and data abstraction.

(Note that some techniques can be closely wedded to a language. For example, the idea of typed data in Pascal shapes the syntax of that language.)

Beyond typed data, there are programming techniques that cannot be made mandatory through imposition of syntax. A familiar example is factoring routines to keep them simple and short. This is not likely to become a part of the imposed syntax of any language. We must use such techniques voluntarily, which is likely to happen only when we develop enough appreciation for them.

Even when one language supports a technique better than others do, the technique still offers benefits to programs written in other languages. This is especially true for Forth, because of its extensibility. We can circulate the benefits pertaining to certain languages throughout all of our own programs. Forth programs can—and should—make use of data typing, object-orientation, and data abstraction, even if Forth itself remains essentially unchanged.

First, a brief examination of Forth constants and variables is offered. These objects are universally available and should already be understood well.

Standard Forth Objects

In the last installment, we learned that an object is a collection of properties. Among those are the layout properties, such as the order of its components. Other, more derivative properties arise because of the consistent interpretations we associate with particular components of an object, such as the sign bit.

While all Forth variables have a sign bit, different host computers use a different bit for the purpose. Likewise, the least-significant eight bits of a 16-bit number may occupy the low byte on some hosts, and the high byte on others.

Note that the same source code should compile and behave the same, whether or not different hosts structure variables in the same way. We have grown comfortable with a certain lack of concern over the host computer, since each Forth implementation treats the bits within a variable in the appropriate way for the given host. But when we extend Forth with new data objects and associated operations, we must give these host-specific concerns their due.

Now that 32-bit processors are becoming commonplace, there is more sensitivity to how variables and constants are implemented. The porting effort for variables and constants will be increasingly difficult. However, much of the code intended for 16-bit variables should work when 32-bit variables are used instead.

If the bit-width property of variables and constants is important for our applications, then we should not use the usual Forth operators. Instead, we should use new compiler directives like L® that can compile the correct operation (@ or D®) for a given host. See Mitch Bradley's discussion in Forth Dimensions (BRA87) for a complete treatment of this subject.

New Objects

New Forth data objects are the primary concern. How can they be designed for ease of porting?

Along with their associated operations, new data objects depend on the host architecture. This is one of the major sources of all porting problems. Among such host peculiarities are bit-processing widths and word-alignment requirements.

Executable routines apart from data structures are more portable, relatively speaking. When kernel routines generate and manipulate addresses, they do so correctly for the host computer. Higher-level routines remain host independent when they engage those kernel routines for their address manipulation needs.

In this way, data objects with simple layouts are often source-code compatible across all hosts. This can happen because these objects avoid address manipulations not performed automatically by the Forth kernel. Examples include variables and constants. In contrast, arrays are a porting problem because of their more expansive layout and the need for a programmer-supplied, element-addressing operation.

But can we really avoid manipulating addresses in high-level code? If not, we should at least try to minimize the adverse effects of address manipulation on portability.

Object-Oriented Techniques

The broadening of our perception of
an object to include its associated operations is the key to object-oriented programming. In the same way that local variables are the private resources of their associated routine, object-oriented operations are subordinate to a given type of object. Accordingly, they are the methods belonging to an object.

(Sharing of operations is possible, however, unlike sharing of local variables. This sharing is restricted to a hierarchical data typing system. So operations are not sharable by all objects, but only those which are a subclass of the superclass for which the methods have been defined. The process of sharing methods down through the ranks of object types is known as inheritance.)

Object-oriented languages treat objects and their associated operations as if they were one entity. Those operations exist only within the context of the object. Within the implementation of the object, those operations are accessible. Within the application, they are referenced via a method-selecting operator or message. For typical Forth purposes, this message sender is an executable routine that leaves a value (message) on the stack for the object to process at run-time. To make a reasonable response, the object must be able to invoke a corresponding method for each message received.

We can parallel these techniques in Forth in many ways (see bibliographic references). One way is to include all the object-specific operations in the DOES> portion of the declarator (HAM86). Each operation can be a separate case in a CASE statement. Such an object will expect a flag (message) on the stack, and will use it to select the correct operation to perform.

By including object-specific operations as part of the declarator, we isolate the porting effort from the reformulation of the object declarator. The message-generation operations will usually port easily, since they can simply be constants.

But a change of syntax has to occur as well. As expected, we lose direct control over which operator is compiled. Instead of specifying the object-specific operation, we now have to send to the object a message indicating what kind of operation is intended. The resulting syntax is prefix (method-selector followed by the object) rather than postfix.

While I consider the prefix syntax a distinct disadvantage of an object-oriented language, there are significant advantages. Eliminating object-operator mismatches and reducing the need for a proliferation of object-specific operators (L8, C8, W8, etc.) are two of these benefits.

Note that object-oriented techniques do more than detect type mismatches — they prevent them! (This is already true for constants, where you never get the opportunity to use C8 or D8 in place of the @ that is compiled inside CONSTANT.)

One correctable problem with object orientation is that it can encapsulate operators in an object so well that they are not reusable by other objects. An inheritance capability is usually offered to counteract this restriction. But why deny reusability, only to permit it again in a restricted form? This strikes me as similar to the debate over enforced data typing, something Forth never seems to have acquired.

To aid our freedom of choice, I will be offering a form of Forth data typing that can be fine-tuned with respect to its restrictiveness. By choosing to overload operators and use well-known vocabulary refinements, it is possible to introduce an approximation of inheritance into this object-oriented data typing scheme. It, too, is adjustable in terms of its restrictiveness: restrictions may be hierarchical (like OOP languages), or there may be no restrictions, or there may be restrictions that fall somewhere between these two extremes.

**Revisiting Ham’s Arrays**

In my opinion, important experiments with object-oriented techniques were kicked off by Michael Ham, even though he makes few claims to object orientation (HAM86). Rather, he applied such techniques in his array declarator. By creating single-element arrays, you will see that Ham already supports the creation and processing of multiple data types, using a single type/object/array declarator. Degenerate cases of arrays can simply be used to create instances of variables, doubles, and anything else, as the following code illustrates (FOR is Ham’s name for his array declarator):

```forth
: VARIABLE
  1 CELL FOR ;
: VARIABLE COUNT

: DOUBLE
  1 DOUBLE FOR ;
: DOUBLE DOLLARS
```

Another advantage is that instances of objects so defined are forced to share certain characteristic properties. For example, each instance of a variable defined in this manner will contain an extra byte value reflecting the array type (necessary to properly index a mixture of array types with one array declarator). With this extra information, Ham can more easily establish a data typing scheme: his operators expect objects like these to contain element-identifying information at the same relative address.

Ham uses overloaded method selectors, such as GET and PUT, to reference the correct fetch and store operation for a given type of object in an array. While this eliminates the need to remember object-specific fetch and store operators, the general loss of a postfix (object-operator) syntax is a dear one. See Terry Rayburn’s FORML paper (RAY87) for an excellent description of the problems that resulted from his trial use of a prefix syntax.

**Data Abstraction Techniques**

Hiding implementation details behind a simpler interface is not a strange Forth technique: it is the Forth programming model (factoring). In Forth, this technique applies to executable routines as well as data objects. For example, WORD has to take into account the layout properties of the text input buffer and the disk block buffer (one may be null-terminated). However, operations that reference WORD need not be concerned with the layout of the buffer. So WORD hides whether input came from the command-line buffer, a disk block buffer or, possibly, a text file.

The declarator that creates constants (CONSTANT) also represents the advantages of abstraction. With a CREATE phrase to determine the layout of the constant and a DOES> phrase to interpret that layout, the declarator CONSTANT encapsulates all the object-specific processing required for constants. In this way, most details of an object’s layout can remain hidden to routines other than the declarator.

Constants are not a good example of data abstraction, however, since they not only interpret the object, they act upon it by moving a value onto the stack.
For Programming Professionals: an expanding family of compatible, high-performance, Forth-83 Standard compilers for microcomputers

For Development: Interactive Forth-83 Interpreter/Compilers
- 16-bit and 32-bit implementations
- Full screen editor and assembler
- Uses standard operating system files
- 400 page manual written in plain English
- Options include software floating point, arithmetic coprocessor support, symbolic debugger, native code compilers, and graphics support

For Applications: Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 8086, 68000, 6502, 8051, 8096, 1802, and 6303
- No license fee or royalty for compiled applications

For Speed: CForth Application Compiler
- Translates "high-level" Forth into in-line, optimized machine code
- Can generate ROMable code

Support Services for registered users:
- Technical Assistance Hotline
- Periodic newsletters and low-cost updates
- Bulletin Board System

Call or write for detailed product information and prices. Consulting and Educational Services available by special arrangement.

The real objective is to hide the implementation (layout properties) of an object behind a simpler interface. The application of this technique also helps reduce the porting effort, as will be shown.

Data abstraction differs from object orientation, because methods need not be considered subordinate to an object. The major work required to abstract an object takes place in the basic operations serving the object. Those basic operations interpret all the peculiarities of the data structure in order to provide simplified access to them. Other operations can use this simplified interface to perform — in a less object-specific manner — the remaining processing of the object.

The operations (or methods) primarily responsible for abstracting objects need to be tightly coupled to the associated objects. Methods that can be shared across objects need to be more loosely coupled to the associated objects.

Naturally, the initial operation is a good place to do the abstraction work, locating it in the DOES> phrase of the object declarator.

Using this approach to create a counted-string data type, we would be obliged to include a COUNT-like operation in the DOES> phrase of the definer, as follows:

```
: COUNTED-STRING
CREATE PAD HERE OVER C@ 1+ DUP ALLOT CMOVE DOES> DUP C@ SWAP 1+ ;
```

Once defined correctly for a given host, this definition produces string variables that can be printed by TYPE without an intervening (host-specific) COUNT operation.

Until we are able to perform all string operations using the object’s external interface, data abstraction is not completely realized. A more wholehearted attempt at abstracting strings would support more string operations, including reliable string concatenation. To do so, the initial operation should hide the object’s layout better. For example, a string concatenation operation should not have to change, even if the maximum-count were to be allotted two bytes rather than one.

Figure One is a more abstract implementation of twice-counted strings. Twice-counted strings include storage for two
counts: the maximum count and the current count. The variable MaxCount provides a simpler interface for accessing the object's maximum count property. This variable can be accessed easily, whereas accessing the actual object's maximum count property requires a string-specific operation. Also shown is Get2CString and Concat2CString. One allows the input of a twice-counted string and the other performs string concatenation between a character array and a twice-counted string.

Note that existing counted-string operations can be shared by twice-counted strings. Such operations include TYPE and -TRAILING.

Once you have verified that TwiceCountedString works properly on a new host, Get2CString is certain to work as well.

However, new syntax dependencies are introduced: Get2CString assumes that the value of MaxCount has been set properly prior to its execution. The way to guarantee this is through a syntax rule: Use Get2CString directly after a reference to a twice-counted string. A similar syntax rule should be applied to Concat2CString.

The introduction of new syntax requirements are part of the problems we often encounter when applying data abstraction and object-oriented techniques. For example, few message-passing, prefix syntaxes allow a consistent version of the following Forth code:

```
CURRENT @ @ - - - > ?
```

At least in most of the current Forth implementations of object-oriented prefix syntaxes (see bibliographic references), GET GET CURRENT will not perform the same function. What does work is:

```
GET CURRENT @
```

although it is an unpleasant mixture of prefix, message-passing syntax and standard postfix syntax.

Other Valuable Abstractions

Although we have discussed data-abstraction techniques useful for hiding the details of an object's implementation, often host peculiarities are also contained in those implementations. We need to deal with those host peculiarities using abstraction techniques once more.

In general, two forms of abstraction are important, data abstraction and host abstraction. To make operations reusable across several objects (or several versions of the same object), abstract objects. To make all data objects work properly across several host computers, abstract host computers.

Since the abstraction of the host can be handled separately, it can and should be solved apart from the abstracting of objects. We should be able to use this code within our applications so they can be transported easily to substantially different hosts.

Conclusions

Although we have forayed deeply into a couple of timely design topics, the discussion has identified the leading factors that interfere with object portability:

1. Host peculiarities, such as bit-processing widths.
2. Object peculiarities that a variety of operations depend on in order to work.

A way to overcome these defects has also been suggested:

1. Hide object-implementation details with data-abstrac-
   tation techniques to minimize the amount of code that needs to change.
2. Identify the implementation details regarding an object that are really host par-
   ticularities, and hide those host particularities under another suite of host-abstrac-
   ting routines.

These two, cascaded layers of abstraction — objects and hosts — should bring about nearly optimal efficiency and portability of data design.

In the next installment, alternative ways of performing address arithmetic are suggested, so that address arithmetic in general is rendered portable. This will help substantially with Forth's abstraction of the host computer.

We also have explored various ways to implement some of the forms that object-oriented extensions to Forth have taken. Progress has been made using multiple code fields (SHA79), placing all the operations for an object within its declarator (HAM79), and operator-lookup tables (RAY79). I favor preserving postfix syntax at any cost, which seems to be the direction of Terry Rayburn as well.

References


Mike Elola is a published Forth programmer and a full-time writer at Apple Computer. Over the years, Mike feels, Forth has tricked him into believing that he is a computer scientist.

```
VARIABLE MaxCount

: TwiceCountedString ( maxbytes -- )
  CREATE C, C+ ( adr count -- )
  DOES>
    DUP C@ MaxCount ! +
    C@ SWAP 1+ ;

: Get2CString ( adr count -- )
  DROP ( the old string length is ignored)
  MaxCount @ MIN EXPECT ;

: Concat2CString ( adr count dest-adr dest-count -- )
  2 PICK OVER + ( totalcount -- )
  MaxCount @ MIN ( validtotalcount -- )
  DUP 3 PICK C! ( C! is object-specific, <> abstraction)
  OVER - ( adr count dest-adr dest-count movecount -- )
  >R + 1+ SWAP DROP R> ( adr dest-adr+ movecount -- )
  CMOVE ( CMOVE is also object-specific, <> abstraction) ;
```

Figure One. 'Abstract' version of twice-counted strings.
Let's talk. Let's really talk about ideas and their presentation. That is the premise behind Real-Time Conferences, or RTCs, on the GENie Forth RoundTable. If you are not participating in these conferences, you are depriving yourself of a unique experience.

The conferences can be broken into three distinctly different categories. The regular Thursday night Figgy Bar is a free forum that resembles a food-fight of ideas. I must confess, I usually am the sysop responsible for these exercises in chaos, though Dennis Ruffer has had his fair share of sitting in as meeting leader. The central theme of these Thursday night (9:30 p.m. Eastern) meetings is anything goes, as long as the discussion involves Forth or items of interest to the Forth community. Such items have included image processing, radar control, robotics, chapter activities, standards, and comp.lang.forth on Usenet.

Sunday night (8:30 p.m. Eastern), Leonard Morgenstern conducts his Question-and-Answer Figgy Bar for intermediate and beginning Forth programmers. His first full conference was on arrays, and I cannot imagine anyone involved with Forth not gaining insight from the presentation. Leonard became a sysop because of his unselfish desire to share his knowledge of Forth, and it is a pleasure to observe him in his natural element. Do not let the stated target group of the Sunday night sessions dissuade you from attending — I can absolutely promise, these are for anyone interested in learning more about Forth.

The third category is our guest conferences. These represent a marvelous opportunity to rub elbows with the movers and shakers of Forth while sitting comfortably at our own keyboards. Guests to date have been Don Colburn (“The Sacred Cows of Forth,” October 1987), Gary Feierbach (“Forth and the Super8,” April 1988), John Hayes and Marty Fraeman of Johns-Hopkins Applied Physics Lab (“FRISC3 32-bit Forth Computer,” May 1988), and Mahlon Kelly (“Forth as a Teaching Tool,” July 1988). By the time this makes print, we should have had a guest conference with Mitch Bradley.

"Thursday night's Figgy Bar resembles a food-fight of ideas."

What wonderful experiences these guest conferences have been. The rest of this column is devoted to the opening comments that started each of them.

Don Colburn
Creative Solutions, Inc.
October 1987

"Let's talk about 16-bitness, and text files and then discuss 'other' categories."

I think Chuck Moore presents a very good case for 16-bitness, for controller applications. A lot of people continue to be confused over the applicability of 16-bit systems on workstation computers. I note that both Alan and Gerry have worked with 32-bit systems. Is this still an issue?

My only point is that I'd like to avoid
having to explain for the 100th-or-so time at the next standards meeting why I find the suggestion that I precede every @ with a .@ or some other silly suggestion [sic]. Glad to see that this may be less of an issue than I've been told it was."

Gary Feierbach  
Inner Access  
April 1988

"Greetings. First I will start with some implementation notes.

"The 8K nucleus contains the following: all good Forth words, including assembler with the entire Super tasker, including all good Forth words, including assembler with the entire Super tasker, including those, along with a Forth-style structured heads! The PC as a file server (BLOCK, etc.) and some 3DUP. Like BEGIN, UNTIL, WHILE, REPEAT, IF, ELSE, THEN. The development ROM contains disk I/O words for using the PC as a file server (BLOCK, etc.) and some other miscellaneous stuff.

"The Super 8 itself runs at 20Mhz, but this is misleading to those not familiar with the flim-flam of chip micro makers: it is immediately divided by 2 on board the chip, and all timings in the manual are related to this 10Mhz clock.

"It is still quite fast, however, about 2.5 times faster than my PC that runs at 4.777Mhz; on the other hand, it isn't a Novix chip so don't get too excited. This is a $7 item to go into automobile dashboards or washing machines. It contains two counter timers, a UART, five eight-bit parallel ports (four are bit-programmable), a DMA channel, vectored interrupts, 277 general-purpose registers, and a partridge in a pear tree.

"The implementation is as close to the F83 public-domain implementation as possible."

John Hayes and Marty Fraeman  
Johns-Hopkins University  
Applied Physics Lab.  
May 1988

"Over the past couple of years, we have designed a number of 32-bit Forth processor chips. We call our chips FRISCs (Forth Reduced Instruction Set Computers). Tonight we want to talk about our latest effort, FRISC 3. To find out more about FRISC 1 and 2, see the 1986 FORMIL Conference Proceedings or the 1987 Rochester proceedings.

"FRISC 3 is a word-addressed machine (i.e., no bytes). All internal elements of the chip are fully 32 bits wide. The top portions of the parameter and return stacks are cached on chip, to improve performance and retain a single path between memory and the CPU.

"The FRISC 3 instruction set consists of eight instructions in three categories:

control flow instructions  
call  
branch  
conditional branch

load/store instructions  
load  
store  
load address low  
load address high

other  
microcode

"All eight instructions take one clock cycle to execute except for load and store, which take two cycles. All FRISC 3 instructions are 32 bits wide. The three instruction formats are shown in Figure One.

"The three msbs of the instructions select one of the instruction types. In the control flow instructions, the remaining 29 bits are an absolute destination address. In load and store instructions, the 16-bit offset is added to R1 to form an address. R2 is the destination of load instructions and the source for stores. The load address instructions make the same address calculation just described, but load the address into R2. In the microcode instruction, there is a 16-bit ALU control field instead of the offset field. The microcode implements most Forth primitives (i.e., DUP, OVER, +, -, etc.). Both the load/store and microcode instructions have a return bit."

Mahlon Kelly  
co-author, Forth: A Text and Reference  
July 1988

"I would like to put forward the idea that Forth is an excellent language for teaching introductory courses in computers. In fact, that it is by far the best language available.

"Most languages were developed for use on mainframes, and although the usual stated reason for their development is to make it easy for humans to use the machines, another is to protect the machines from humans. Thus, those languages are designed to prevent users from being aware of the equipment itself. That stands in the way of students really understanding how computers work.

"Forth does the opposite. The more you know about the machine, the better you can use Forth. Thus, the student is encouraged to learn not only about the language, but about computers. And since Forth is interactive, Forth encourages the best type of learning, that is, learning from mistakes.

"Direct memory and register access are both available, and perhaps more important is the use of any desired number base. It is very easy to teach about binary, octal, decimal, and hex arithmetic. It is very easy to teach about logical operators. And it is even very easy to introduce assembly language.

"Perhaps most importantly, it is easy to teach about machine and assembly instructions, and to get across the differences among machine, assembler, and higher-level languages. In the first lecture of my course, I teach how to define SQUARE DUP ;

"Then I have them look at and disassemble DUP and *. Now when they program, they know what they are doing to the machine. They do this while sitting at computers.

"I have been told by one M.S. student in computer science that after my Forth course (which he obviously took to learn the language, not computers), he had the equivalent level of understanding of computers of a third-year student in comp sci. The course assumes no background.

"Questions: Are these ideas correct, or should we continue to teach introductory Pascal, BASIC, Fortran, etc.? How is Forth best taught — is it best to teach first how it works, or how to use it? How can we convince computer science departments of Forth's value, not only as a language but as a teaching tool?"

Hope to see you at our next Real-Time Conference where we can talk — really talk — about Forth.

(Figure One on next page)
(Screens continued from page 25.)

(Figure One. FRISC 3's three instruction formats.)

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

U.S.A.

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

- **ALASKA**
  - Kodak Area Chapter
    Horace Simmons
    (907) 486-5049

- **ARIZONA**
  - Phoenix Chapter
    4th Thurs., 7:30 p.m.
    AZ State University
    Memorial Union, 2nd floor
    Dennis L. Wilson
    (602) 956-7578

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

- **CALIFORNIA**
  - Los Angeles Chapter
    4th Sat., 10 a.m.
    Hawthorne Public Library
    12700 S. Grevillea Ave.
    Phillip Wasson
    (213) 649-1428
  - North Bay Chapter
    2nd Sat., 10 a.m. Fortu, Al
    12 Noon Tutorial, 1 p.m. Fortu
    South Berkeley Public Library
    George Shaw (415) 276-5953
  - Orange County Chapter
    4th Wed., 7 p.m.
    Fullerton Savings
    Huntington Beach
    Noshir Jesung (714) 842-3032
  - San Diego Chapter
    Thursdays, 12 Noon
    Guy Kelly (619) 454-1307
  - Sacramento Chapter
    4th Wed., 7 p.m.
    1708-59th St., Room A
    Tom Ghormley
    (916) 444-7775
  - Silicon Valley Chapter
    4th Sat., 10 a.m.
    H-P Cupertino
    Bob Barr (408) 435-1616
  - Stockton Chapter
    Doug Dillon (209) 931-2448

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

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

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

- **GEORGIA**
  - Atlanta Chapter
    3rd Tues., 6:30 p.m.
    Western Sizzlen, Doraville
    Nick Hennenfent
    (404) 393-3010
  - Central Illinois Chapter
    Champaign
    Robert Illyes (217) 359-6039

- **ILLINOIS**
  - Cache Forth Chapter
    Oak Park
    Clyde W. Phillips, Jr.
    (312) 386-3147
  - Central Iowa FIG Chapter
    1st Tues., 7:30 p.m.
    Iowa State Univ., 214 Comp. Sci.
    Linus Orth (913) 236-9189
  - Fort Wayne Chapter
    2nd Tues., 7 p.m.
    IP Univ. Campus, B71 Neff Hall
    Blair MacDermid (219) 749-2042
  - Illinois
    Cache Forth Chapter
    Oak Park
    Clyde W. Phillips, Jr.
    (312) 386-3147

- **INDIANA**
  - Fort Wayne Chapter
    2nd Tues., 7 p.m.
    IP Univ. Campus, B71 Neff Hall
    Blair MacDermid (219) 749-2042
  - Central Iowa FIG Chapter
    1st Tues., 7:30 p.m.
    Iowa State Univ., 214 Comp. Sci.
    Rodrick Eldridge (515) 294-5569

- **IOWA**
  - Central Iowa FIG Chapter
    1st Tues., 7:30 p.m.
    Iowa State Univ., 214 Comp. Sci.
    Rodrick Eldridge (515) 294-5569

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

- **MICHIGAN**
  - Detroit/Ann Arbor Area
    4th Thurs.
    Tom Chrapkiewicz
    (313) 322-7862

- **MINNESOTA**
  - MNFIG Chapter
    Minneapolis
    Even Month, 1st Mon., 7:30 p.m.
    Odd Month, 1st Sat., 9:30 a.m.
    Fred Olson (612) 588-9532
    NC Forth BBS (612) 483-6711

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

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

- **NEW MEXICO**
  - Albuquerque Chapter
    1st Thurs., 7:30 p.m.
    Physics & Astronomy Bidg.
    Univ. of New Mexico
    Jon Bryan (505) 298-3292
**NEW YORK**
FIG, New York
2nd Wed., 7:45 p.m.
Manhattan
Ron Martinez (212) 866-1157

Rochester Chapter
Monroe Comm. College
Bldg. 7, Rm. 102
Frank Lanzafame
(716) 482-3398

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

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

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

**TENNESSEE**
East Tennessee Chapter
Oak Ridge
2nd Tues., 7:30 p.m.
800 Oak Ridge Turnpike
Richard Seerist
(615) 483-7242

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

Houston Chapter
3rd Mon., 7:45 p.m.
Intro Class 6:30 p.m.
Univ. at St. Thomas
Russell Harris (713) 461-1618

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

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

**Potomac FIG**
Penton Fig
Arlington
1st Tues.
Lee Recreation Center
Joel Shprenz
11918 Winterthur Ln., #107
Reston, VA 22091

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

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

MAD Apple Chapter
Bill Horton
502 Atlas Ave.
Madison, WI 53714

**INTERNATIONAL**
**AUSTRALIA**
Melbourne Chapter
1st Fri., 8 p.m.
Lance Collins
65 Martin Road
Glen Iris, Victoria 3146
03/929-2600
BBS: 61 3 299 1787

Sydney Chapter
2nd Fri., 7 p.m.
John Goodsell Bldg., RM LG19
Univ. of New South Wales
Peter Tregoeagle
10 Binda Rd., Yowie Bay
2228
02/524-7490

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

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

**CANADA**
BC FIG
1st Thurs., 7:30 p.m.
BCTT, 3700 Willingdon Ave.
BBY, Rm. 1A-334
Jack W. Brown (604) 596-9764
BBS (604) 434-5886

Northern Alberta Chapter
4th Sat., 1 p.m.
N. Alta. Inst. of Tech.
Tony Van Muyden
(403) 962-2203

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

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

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

**ITALY**
FIG Italia
Dr. Marco Tausel
Via Gerolamo Forini 48
20161 Milano
02/435249

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

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

**SPECIAL GROUPS**
**NC4000 Users Group**
John Carpenter
(415) 960-1256

**REPUBLIC OF CHINA**
(R.O.C.)
Ching-Tang Tzeng
PO Box 28
Lung-Tan, Taiwan 325

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

**SWITZERLAND**
Swiss Chapter
Max Hugelshofer
Industrieberatung
Zibberstrasse 6
8152 Opfikon
01 810 9289
FORML CONFERENCE

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

Following Thanksgiving, November 25–27, 1988

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

Forth and Artificial Intelligence

Artificial intelligence applications are currently showing great promise when developers focus on easy-to-use software that doesn’t require specialized expensive computers. Forth’s design allows programmers to modify the Forth language to support the unique needs of artificial intelligence.

Papers are invited that address relevant issues. Papers about other Forth topics are also welcome.

Mail your abstract(s) of 100 words or less to FORML Conference, Forth Interest Group, P.O. Box 8231, San Jose, CA 95155.

Completed papers are due November 1, 1988.

Conference Registration

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

Conference attendee in double room - $275 • Non-conference guest in same room - $150 • Children under 17 in same room - $100 • Infants under 2 years old in same room - free • Conference attendee in single room - $325

Register by calling the Forth Interest Group business office at (408) 277-0668 or writing to: FORML Conference, Forth Interest Group, P.O. Box 8231, San Jose, CA 95155.