VOLUME XI, NUMBER 1 MAY/JUNE 1989

FORTH DIMENSIONS

FORTH DIMENSIONS

EDITORIAL

Forth Dimensions
Published by the
Forth Interest Group
Volume XI, Number 1
May/June 1989
Editor
Marlin Ouverson
Advertising Manager
Kent Safford
Design and Production
Berglund Graphics

Welcome to the eleventh volume of *Forth Dimensions*. As a member of the Forth Interest Group, you are part of a diverse audience of readers. A beginning Forth programmer will find here the fellowship of other novices, but in a 1987 survey 78% of our readers rated their knowledge of Forth as intermediate or advanced. Our authors are drawn from among them, so these pages offer exposure to both traditional and ground-breaking (or rule-breaking) ways of approaching various programming problems. Differing opinions are printed, letters with feedback about previous articles are welcome, and improvements to earlier code are published regularly. Rather than dictate one right approach to a problem and censor the rest, we try to keep this forum open for the cross-pollination and evolution of ideas. If the debate sometimes seems to spiral dangerously close to the sun, that is one side effect of working with a language in which the language itself is easily subject to modification. Just follow where your Forth system leads: it will prod you into understanding its low-level details, and even your own computer hardware, better than you had planned.

A stumbling block is sometimes encountered when you attempt to use Forth code from another system and find that it won't run (it may even refuse to load) after you bring it into your system. A Forth system generally belongs to one of several broad categories. It may conform to the Forth-83 or Forth-79 Standard; to fig-FORTH, F83, FPC, or another public-domain version; or to a vendor's unique way of implementing the language. Our articles indicate which dialect is used by the authors but, within those descriptive categories, assembler-specific code may affect the portability between machines, and vendors have introduced some differences (e.g., features) even between brand names that are based on the same standard.

The trick is to turn every would-be obstacle into an opportunity to become more expert about how your system works. You will learn which words are implemented elsewhere under other names and which really are functions unique to your system. Eventually, you will understand its strengths and weaknesses in different environments, in relation both to other Forths and to non-Forth systems. By that time, we hope you will have joined the ranks of our authors to share your experience, to write about your latest project, and to advance the state of our collective understanding.

* * *

Local variables: in the past they've been either ho-hum ("my system has had them for years") or taboo ("not Forth-like"), depending on who you asked. Like many formerly acrid debates, the arguments have faded gradually with the popular endorsement of fully featured programming environments. Now not just a Forth-too issue, local variables have become a convenience many programmers simply don't want to do without. Add the ability name them, and they become more intuitive and readable than the stack-juggling alternatives.

Those who want to seriously study local variables should look in *A Bibliography of Forth References*. Several related articles by prominent Forth thinkers have been published and still lend themselves to discussion of the topic. Three articles in this issue address the subject, too. Depending on your Forth, your programming style, and your view of the tradeoffs, one or the

(Continued on page 41.)

About the Forth Interest Group

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

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $24/36 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 8231, San Jose, CA 95155."

LETTERS

} Serial Dates for FPC
Dear Editor,

After submitting my article on a suggested alternative for writing Forth programs (*Forth Dimensions* X/6), I find that a short addition to FPC will allow the source code for "A Serial Day Calculation" to compile with that implementation, too.

```
{
: }  ( -- )
\ skip all lines between } and {
  ?loading
  [compile]
  " (" ">$ comment$ !
  ['] <comment:> is run ;
}
```

I hope this will encourage Forth programmers to give the technique a try. Even this letter is written in the style!
Yours,
Glen B. Haydon
Box 429, Route 2
La Honda, California 94020
{

Objects are Harmonious
Dear Mr. Ouverson:

This is a comment on Mike Elola's article (*Forth Dimensions* X/5) about object-oriented Forth (OOF). I have written an object-oriented Forth interpreter and can respond to many of Mr. Elola's points and contribute some more.

The object-oriented approach

There is no doubt the object-oriented approach is the best design of data structures that has come about. However, there is much more to the object-oriented approach than data structures; it is a software design methodology. This methodology views the world as a collection of objects. Each object has certain properties, called instance variables; and there are particular operations that can be performed on these variables, called methods.

Objects communicate with each other by sending "messages" to one another. The object management system has the duty to see if a message matches any of the methods that the object receiving the message understands. If it does, it invokes that method within the object; otherwise, the system returns an error.

This approach encapsulates everything about the object in its very body, which simplifies maintenance and debugging and reduces the probability of error. Forth, on the other hand, views the world as a collection of words, each doing its own thing and communicating with each other through the stack.

Classes of objects

Every object belongs to a "class." A class is merely a template that describes the instance variables and methods that objects of the class will have when they are created. For example, a string class might define the variable `Length` and the methods `Concat`, `Insert`, `Delete`, and `Substr`. The words `STRING ST` will create an object `ST` of class `STRING` much like any defining word. `ST` will show up in the dictionary as a regular Forth word.

The words `ST LENGTH` will send the message `LENGTH` to the object `ST`. The object management system will fetch `ST` for the `LENGTH` method. This process is called "binding." Contrary to what Mr. Elola said, message-passing to objects can use postfix notation, as in this example. However, there is nothing wrong with prefix syntax, since Forth uses it all the time with every defining word (such as `VARIABLE` and `VOCABULARY`). Binding in the example will succeed and the method `LENGTH` will be executed, pushing the current length of `ST` on the stack.

This example shows that incorporating classes and objects in Forth is almost natural. In my Forth interpreter, called CSU Forth, I defined the two words `:CLASS` and `;CLASS` to start and end the definition of a class. Between these two words, methods and variables are defined in the regular Forth manner, e.g., using : and ; as in Figure One, which shows an example definition of a stack class.

Inheritance

The word `INHERIT` in the stack example is used to establish any number of "parents" of the class being defined. By doing so, the system—when it cannot bind a message to an object method—will look in the object's parents for it. This way, objects can be reusable and extensible. `0 INHERIT` means this class does not inherit any methods or instance variables from any other class. Such a class is said to be original.

Mr. Elola emphasized that inheritance, particularly multiple inheritance in which a class inherits from more than one parent, is what makes the object-oriented design so attractive. However, he said, it is undisciplined and requires care and patience. I don't exactly know what he meant by that, but I suppose that one of the things to watch out for is methods with identical names in the different parent classes (name collisions). Which one should the system bind to?

```
:CLASS STACK  0 INHERIT
   VARIABLE TOS   VARIABLE BOS    ( top and bottom of stack)
   : CLEAR       BOS @ 1+ TOS ! ;
   : INITIALIZE  BOS !  CLEAR ;  ( BOS needed to create object)
   : DEPTH       BOS @  TOS @  - 1+ ;
   : -EMPTY      DEPTH 0> ;
   : SHOW        -EMPTY IF
                   ASCII ( EMIT
                   TOS @ 1- BOS @ DO I @ . -1 +LOOP
                   ASCII ) EMIT
                 ELSE ." ( )" THEN ;
   : POP         -EMPTY IF 1 TOS +! THEN ;
   : +STACK      -1 TOS +! ;  PRIVATE ( inaccessible outside class)
   : PUSH        +STACK TOS @ ! ;
;CLASS
```

Figure One. Definition of a stack class.

```
:CLASS FIXED-STACK  1 INHERIT STACK
   VARIABLE MAXSIZE     ( max num of elements)
   : INITIALIZE  BOS !  MAXSIZE !  CLEAR ; ( 2 parms to create obj)
   : -FULL       DEPTH MAXSIZE @ < ;
   : +STACK      -FULL IF -1 TOS +! THEN ;  PRIVATE
;CLASS
( Notice that INITIALIZE and +STACK are redefined, thus overriding
  old definitions in parent STACK; however PUSH which uses +STACK
  was not! That's the power and elegance of object-oriented design)
```

Figure Two. A new definition can inherit features from a class.

I solved this problem by establishing the order of classes in the inheritance list as the order of the search for methods. Another technique that has been used is to rename the methods. Figure Two shows the definition of a fixed-size stack that inherits many features from the class STACK.

Obstacles to OOF

Under this subtitle, Mr. Elola mentioned the need for more than one data stack to be able to handle the variety of classes, therefore OOF has to "steer clear from normal Forth." There is nothing in the standard that limits a Forth implementation from having more than one data stack. An implementation can employ several stacks, as long as their details are hidden and do not interfere with the data stack. The best way to do this is through classes.

Under the subhead "I Object!" Mr. Elola gave an example of the need to have prefix notation to implement OOF. His example was attempting to calculate the average of two numbers, as follows:

```
GET A
GET B
+ 2/
PUT C
```

There is no reason why this problem should be designed that way. It's a bad design. Here is how CSU Forth solves the same problem:

```
USE FLOAT
A @
B @
+ 2/
C !
```

The words USE FLOAT establish that the search order for methods will start with the FLOAT class, much like the vocabulary search order. The syntax is postfix and there is no binding ambiguity. The words @, +, 2/, and ! are all defined in the class FLOAT and A, B, and C are objects of it. No compromises on Forth's original philosophy are needed. In my experience, there are *no* obstacles to implementing OOF.

Why not vocabularies?

It may seem that vocabularies can do everything classes can. Use CHAIN to establish inheritance; to pass messages, use the vocabulary name to alter the search order; and to make objects of a class, use special definitions of CREATE ... DOES>. It's possible, but tedious. It is also a lot better, as all of us know, to implement these mechanisms in assembly rather than in high-level Forth. My experience with CSU Forth convinced me that implementing OOF is easy and harmonious with regular Forth, and doesn't have to interfere with it. CSU Forth has classes *and* vocabularies.

Dr. Ayman Abu-Mostafa
7932 Lampson Avenue #25
Garden Grove, California 92641-4147

Fast */ for the Novix

Dear Sir,

One of the promises of the Novix processor's Math Steps opcodes, in particular *' and /', is that hand-tailored math routines can be written to provide very fast, limited-range, multiply and divide routines.

I can report that this idea does not work for Forth's * or / operations. Performing the Math Steps opcodes fewer than the full 16 TIMES leaves the result left-shifted (for *) or right-shifted (for /) by as many bits as the opcodes were run fewer than sixteen times. For example, 2 3 * gives twelve when *' is run once less; and 100 2 / gives 25 when /' is run once less.

However, it is possible to write short versions of Forth's scaling function */. This is due to the fact that the two operations complement each other, as they rotate the intermediate results in opposite directions. This means that the partial result left by a short multiply can be fed into a short divide routine and the net result is scaled correctly.

In the following definition, n may take any integer value less than 15 without affecting the value of the result, save for different argument ranges.

```
: */' ( u u* u/ -- u')
  >R 4 I!
  0  n TIMES *'
  R> 4 I!
  D2*  n TIMES /'
  DROP  ; ( 12 + 2n )
```

From the timing calculation, it can be seen that an eight-bit */' will take a mere 28 clock cycles, with a 12-bit version consuming 36 cycles, compared to 59 for the full */ provided in cmFORTH.

On the subject of argument ranges, I thought the multiplicand and divisor would be limited to n number of bits, but it transpires that the input number u is also limited to this size.

Yours faithfully,
Dave Edwards, Dir. of Engineering
Jarrah Computers Pty. Ltd.
Suite 7, 85 Rokeby Road
Subiaco, WA 6008
Australia

YES, THERE IS A BETTER WAY
A FORTH THAT ACTUALLY
DELIVERS ON THE PROMISE

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 fool 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 resize segments anytime, without a system regen. With the GigaForth option, your programs transparently enter native mode and expand into 16 Meg extended memory or a gigabyte of virtual, and run almost as fast as in real mode.

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 nested 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 INTCALL 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 blocked, sequential or random I/O. The character stream can be received from/sent to console, file, memory, printer or com port. 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 formatting counterparts for building records. We also provide words to parse all data types with your choice of field definition. HS/FORTH parses files from any language. Other words treat files like memory, nn@H and nn!H read or write from/to a handle (file or device) as fast as possible. For advanced file support, HS/FORTH easily links to BTRIEVE, etc.

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

HS/FORTH provides hardware/software floating point, including trig and transcendentals. 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 RADIANS or DEGREES mode. No coprocessor? No problem. Operators (mostly fast machine code) and parse/format words cover numbers 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 @ - X @ Y @ + / Z ! (26 bytes) Arrays can ignore 64k boundaries. Words use SYNONYMs for data type independence. HS/FORTH can even prompt the user for retry 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 multitasker provides individual user stack segments as well as user tables. Control passes 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 shell 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 reuseable 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, dynamic strings, multitasker RSDL linker, physical screens	$395.
Production	ROM, Metacompiler, Metawindows	$495.
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

Forth-83

EXTENDED BYTE DUMP

ALLEN ANWAY - SUPERIOR, WISCONSIN

Most languages, including Forth, provide a byte dump of RAM memory. The display usually shows data bytes on the left and ASCII text, separately, to the right. In Forth, the programmer would call:
`DUMP ( addr #bytes -- )`

The chief disadvantages of this scheme are that it is cumbersome to browse through memory, and it is necessary to count over on the screen to relate displayed bytes to their ASCII text. So, for several years, I have used this video byte dump:
`VDUMP ( addr -- )`

It lists the ASCII text underneath the bytes (Figure One), and zooms forward through memory with each press of the space bar and backward with the return key. The extreme left column of Figure One displays the 16-bit address in two bytes, and the rest displays the data bytes with their corresponding ASCII characters under them.

Many times I have yearned to move my cursor up to the display bytes and type over them. In Forth, if you want it, you write it; so I present:
`VVDUMP ( addr -- )`

After the display fills the screen, press the up-arrow or escape key, and you can move the cursor to any data bytes and type over them, automatically placing the new values in RAM. Typing characters with the cursor anywhere else will not be recognized by the computer, except the cursor moves right. Press escape again to exit altogether. The space bar and return still browse forward or backward through memory.

Requirements and Modifications

To write and run this program, you need the following words or their equivalents:

`XKEY ( -- ASCII )`
Sees all keystrokes alike, including escape. I don't use my `KEY` because escape doesn't work the same as the other keys.

`-LF, LF, -BS,` and `BS ( -- )`
These move the cursor up, down, right, and left one position on the screen.

> ***"VVDUMP starts the real programming fun."***

`CLEAR ( -- )`
Clears the screen and positions the cursor at the upper left-hand corner before printing.

```
/C
( n -- low_byte high_byte )
```
This is shown on screen 39 if you don't have it.

You do not need the next two words, but it makes a slicker ending with them:
```
HVTAB
( horiz_index vert_index -- )
```
Positions cursor.

`CEOL ( -- )`
Clears to the end-of-line.

In the original `VDUMP`, I wrote my word `LAYOUT` in machine language to print the display fast, but here I show everything in high-level Forth for more general demonstration of the principles.

I also generalized the program for this discussion by writing `HPOS` and `VPOS` variables for the cursor position—in case you don't have access to the internal registers. You can save programming steps if you can gain access to them: on the Apple II computers, they are at $0/0024 and $0/0025, respectively. ($ means 'hex' for Apple people and some others).

Evolution of the Dump

Let's start with `VDUMP`. Step one is to plan the display grid. I don't like planning, but here we are stuck with it. From Figure One (the desired output), we derive the grid in Figure Two showing the coordinate pairs where each nybble will be printed. By this scheme, the 24 x 40 Apple screen allows $B bytes down by $C bytes wide; $B x $C = $84 bytes displayed. Thus, going backward with the return key makes us subtract 2 x $84 = $108 from the ending address to get the new beginning address for the display. (Better Forth programming would use constants for $B, $C, $84, and $108.)

`VDUMP` looks for the keys you press, controlling forward and backward browsing, or exit.

`PDUMP ( start_addr end_addr -- )`
Prints the same format on a printer, but 32 squeezed data bytes wide, to almost fill the width of the paper page.

Working on `VVDUMP` starts the real programming fun. To type over the nybble of a data byte, we use the cursor grid (Fig-

```
09  86 42 52 41 4E 43 48 92 09 B2 0C 85
 89     B  R  A  N  C  H
09  0C 4C D8 08 23 09 87 3F 42 52 41 4E
 95     L        #        ?  B  R  A  N
09  43 48 A5 09 B5 00 08 E8 E8 28 F0 E5
 A1  C  H                    (
09  E6 0C E6 0C 4C D8 08 89 09 86 28 4C
 AD                 L                 (  L
09  4F 4F 50 29 BF 09 18 A9 01 00 63 01
 B9  O  O  P  )                    c
09  83 01 43 03 D0 C7 A0 00 F0 89 B6 09
 C5        C
09  87 28 2B 4C 4F 4F 50 29 DB 09 B5 00
 D1     (  +  L  O  O  P  )
09  E8 E8 85 10 38 A3 03 E3 01 85 12 18
 DD                 8
09  A3 01 65 10 83 01 38 A3 03 E3 01 F0
 E9        e              8
09  D5 45 12 10 98 A5 10 45 12 10 CB 30
 F5     E                 E              0
0A  90 9B 09 84 28 44 4F 29 0B 0A B2 0C
 01                 (  D  O  )
```

Figure One. VDUMP shows ASCII text below each row of hex bytes.

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,25	0,26	0,27
0	9			8	6		8	5	
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,25	1,26	1,27
	8	9							
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,25	2,26	2,27
0	9			0	C		4	E	
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,25	3,26	3,27
	9	5						N	
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,25	4,26	4,27
0	9			E	6		4	C	
12,0	12,1	12,2	12,3	12,4	12,5	12,6	12,25	12,26	12,27
0	9			D	5		3	0	
13,0	13,1	13,2	13,3	13,4	13,5	13,6	13,25	13,26	13,27
	F	5						0	
14,0	14,1	14,2	14,3	14,4	14,5	14,6	14,25	14,26	14,27
0	A			9	0		0	C	
15,0	15,1	15,2	15,3	15,4	15,5	15,6	15,25	15,26	15,27
	0	I							
16,0	16,1	16,2	16,3	16,4	16,5	16,6	16,25	16,26	16,27

Figure Two. Grid of video display (hex numbers). *(Text, continued on page 12.)*

```
    SCREEN # 033
( Original VDUMP in high level Forth )
HEX
: .2N S>D <# # # #> TYPE ;     ( n --- )

: LAYOUT CLEAR
  BEGIN ROT ROT DUP /C .2N >R SPACE
    OVER 0 DO SPACE DUP I + C@ .2N
           LOOP
    CR SPACE R> .2N
    OVER 0 DO SPACE SPACE DUP I + C@
              DUP DUP BL < SWAP 2E > OR
              IF DROP BL THEN EMIT
           LOOP
    CR OVER + ROT 1- ?DUP 0=
  UNTIL ;    ( #cols\address\#rows ---
               #cols\address' )
: VDUMP BASE @ SWAP HEX C SWAP
  BEGIN B LAYOUT XKEY DUP D =
        IF   DROP 108 -
        ELSE BL -          ( space bar )
          IF 2DROP BASE ! EXIT THEN
        THEN
  REPEAT ;           ( ram-address --- )
-->
```

```
    SCREEN # 034
( Original PDUMP in high level Forth )

            ( addr-begin\addr-end --- )

: PDUMP >R >R ?STACK BASE @ HEX
     20 R> R> OVER - 0 20 UM/ 1+
      1 OUTPUT!        ( start printer )
     1B EMIT 71 EMIT CR  ( <Esc>q<Cr> )
  ( squished chars, Apple Imagewriter )
     LAYOUT
     1B EMIT 63 EMIT CR  ( <Esc>c<Cr> )
                        ( reset printer )
      0 OUTPUT!         ( stop  printer )
     2DROP BASE ! ;

-->

Allen Anway                December 12, 1988
1219 North 21st Street
Superior, WI 54880
715-394-4061
```

```
    SCREEN # 035
( New VVDUMP types over data readout )

VARIABLE HPOS     VARIABLE VPOS

: LEGALV VPOS @         DUP 14 <=
     SWAP    2 MOD 0= AND ;
                          ( --- flag )

: LEGALH HPOS @ SWAP - DUP    0>
     SWAP 1- 3 MOD 0= AND ;
          ( offset-position --- flag )

: LEGAL? LEGALV 3 LEGALH 4 LEGALH
     OR AND ;             ( --- flag )

: INDex OVER 84 - VPOS @ 2/  C *
                 HPOS @ 4 - 3 / + + ;
  ( end-ad\nyb --- end-ad\nyb\byte-ad )

-->
INDex calculates ram location that the
  typed-over byte exists, byte-ad.
  C B/* is 84, all in HEX.
  nyb is the nybble entered by DIGIT'.
```

```
    SCREEN # 036
( New VVDUMP types over data readout )

: >>      VPOS @ 15 <=          ( --- )
   IF     HPOS @ 27 >=
     IF   -1 HPOS !  1 VPOS +!
     THEN  1 HPOS +!                -BS
   THEN ;

: <<      HPOS @    0=          ( --- )
     IF   28 HPOS ! -1 VPOS +!
     THEN -1 HPOS +!                 BS
          VPOS @ 0< IF 0 VPOS ! THEN ;

: ^^      VPOS @                ( --- )
     IF             -1 VPOS +!     -LF
     THEN ;

: vv      VPOS @ 14 <=          ( --- )
     IF              1 VPOS +!      LF
     THEN ;

: EMIT' DUP BL  < IF DROP BL THEN
      EMIT BS >> ;        ( char --- )
-->
```

```
    SCREEN # 037
( New VVDUMP types over data readout )
HEX
VARIABLE MODE

: DIGIT'          ( addr\key --- addr )
   MODE @   0= IF  DROP    EXIT THEN
   LEGAL?   0= IF  DROP >> EXIT THEN
   DUP >R
   10 DIGIT 0= IF RDROP >> EXIT THEN
   INDex C@ SWAP 3 LEGALH
   IF     4 SHIFT SWAP 0F AND OR
          DUP vv >> EMIT' ^^ << <<
   ELSE           SWAP F0 AND OR
          DUP vv    EMIT' ^^ <<
   THEN
   INDex C!     R> EMIT' ;

-->

DIGIT' types representative character
   below the byte, then updates nybble.

If you don't have RDROP replace it above
          with R> D
```

```
    SCREEN # 038
( New VVDUMP types over data readout )

-->     commentary for DIGIT'

: DIGIT'          ( addr\key--- addr )
MODE OFF ?        yes:            quit
illegal position ? yes: move curs, quit

illegal digit ?    yes: move curs, quit
get byte, is it at lhs nybble position ?
  yes: shove over 4 bits, insert in byte
       print over display character.
  no:                     insert in byte
       print over display character.

update ram with new byte, print nybble.

-->
```

```
    SCREEN # 039
( New VVDUMP types over data readout )

: .ENTER ." ENTER" -LF -LF BS
         4 HPOS ! 14 VPOS ! -1 MODE ! ;
                                ( --- )

VARIABLE OLDKEY

: XKEY' XKEY DUP OLDKEY ! ;
                             ( --- key )

: EXIT' 2DROP BASE ! 0 16 HVTAB CEOL ;
                        ( base\x\x --- )

: LAYOUT' CLEAR B LAYOUT MODE OFF ;
         ( #cols\addr --- #cols\addr' )

-->

If you don't have /C use this one:
HEX
: /C DUP 00FF AND SWAP FF00 AND
           -8 SHIFT ;
1234 /C . . prints out 12 34
```

```
    SCREEN # 040
( New VVDUMP types over data readout )

( ram-addr --- )
: VVDUMP BASE @ SWAP HEX C SWAP
                 LAYOUT'
  BEGIN XKEY'                CASE
     D OF 108 - LAYOUT'           ENDOF
    BL OF       LAYOUT'           ENDOF
    MODE @              AND
     8 OF <<                     ENDOF
    15 OF >>                     ENDOF
     A OF vv                     ENDOF
     B OF ^^                     ENDOF
    1B OF EXIT' EXIT             ENDOF
    DIGIT' ( removes key )
    MODE @ 0= OLDKEY @ AND
     B OF .ENTER                 ENDOF
    1B OF .ENTER                 ENDOF
    MODE @ 0= IF DROP EXIT' EXIT
             THEN             ENDCASE
  REPEAT ;

DECIMAL ;S
```

(Screens continued on page 40.)

ure Two) to determine (a) is the cursor on an actual data byte, (b) which nybble is it on, and (c) what is that byte's absolute RAM address, so we can update it?

We save our sanity by breaking this complex problem into small pieces. When the cursor is actually on a displayed data byte, I call it a legal position for updating that byte. The vertical spacing is easy: LEGALV checks for odd (illegal) and even (legal) rows. The horizontal spacing is harder: the extreme left address bytes are illegal—they aren't data bytes. So we write LEGALH requiring an offset input of three or four to compensate for starting irregularly.

Combining the two gives us:

```
LEGAL?  ( -- flag )
```

Returns a true flag indicating that the cursor is on a displayed data byte.

Using similar techniques, we figure out INDex to determine the absolute RAM address by calculating from the horizontal and vertical positions and the ending address.

Screen 36 shows cursor movement words for four directions, keeping the cursor from going too low and scrolling the screen. This programming assumes that -LF and BS are unable to scroll the screen backward even if you tried to, which is true of most computers. As stated above, these words are easier to write if you have access to the internal horizontal and vertical indices of the computer.

DIGIT' takes your keyed input and types over the nybble and its corresponding ASCII character. It is defined on screen 37 and explained in screen 38. Screen 39 shows simple utility words, good practice in Forth to clean up the final screen. The word .ENTER assumes a known starting cursor position at grid 16,0 (Figure Two). After printing ENTER in reverse video, the cursor is placed on grid 14,4 at the high nybble of the bottom-most data number.

For your amusement, I include screen 41 to show my first attempt to force this program to work, something like interior decorating with an ax. (It does work!) That code is impossible to understand or to alter successfully, even slightly; and it mixes dissimilar functions. Note also two unstructured branches for complete—rather than partial—obfuscation.

How would one clean up screen 41? Start over with a new concept, that of a MODE variable. If the mode is false, you are browsing; if the mode is true, you are moving the cursor and, maybe, typing over. MODE separates the final VVDUMP program (screen 40) into four easy-to-comprehend sections:

- The space bar and return key browse to the next or previous memory display, turning MODE off (no type-over).
- If the mode is true (-1), the commands <<, >>, vv, and ^^ will move the cursor; and a numerical key will type over, via DIGIT'. The escape key here will exit.
- If the mode is false (0), one can activate type-over with the up-arrow or escape key (running the .ENTER command). This must follow the true mode, where the escape key acts in a contradictory way.

(Continued on page 40.)

F83

LOCAL VARIABLES AND ARGUMENTS

JYRKI YLI-NOKARI - TAMPERE, FINLAND

I will describe a technique to add local variables and arguments (procedural) to Forth. The code is written in Laxen and Perry's F83, but should be quite portable to any conventional Forth implementation. I will not argue about why everyone from now on should use local variables and arguments. If you have been forced to use PICK, you'll find this article at least of academic interest.

Design Goals

My major design goal was to have local variables and arguments that were "clean" and Forth-like. That is, they had to be portable, ROM-able, recursive, and—hopefully—not state-smart. They should also be easy and intuitive to use. Just a declaration at the beginning of a definition should suffice. One important criterion was that the whole source code should be one or two screens total; otherwise, nobody would bother to try it. Local variables and arguments should work similarly, differing mainly in that arguments are like constants and local variables are like variables. This analogy calls for corresponding defining words.

Specifications

A local variable is a variable created at run time by allocating space from the return stack. Its lifetime is from the beginning to the end of the definition in which it is declared, at which time the space is automatically released. Since the space is allocated from the stack, recursion is allowed and code ROM-ability isn't affected. Local variables are accessed by address, like any variable.

An argument is one of the stack inputs a word gets. When arguments are declared, they are moved from the stack to the return stack. (Local variables are similar, but are not initialized.) Arguments are accessed by value, like constants. In the current implementation, they can also be accessed by address, like variables, although this seems to be unnecessary because they are only stack inputs.

"Locals are implemented by using return stack frames."

Programmer's View

`ARGS ( arg1 ... argN N -- )`
declares N stack items as arguments. The stack items arg1 ... argN are pushed to the return stack. When the word that calls ARGS exits, the space is automatically freed.

The general word to push the value of argument N to the stack is:

```
% ( N -- argN )
```

Numbering starts from the deepest stack item (in a stack diagram, the leftmost item). The name % was chosen because it has no previous meaning in Forth, and because of its correspondence to MS-DOS file.BAT parameters.

For fast access, I have predefined the words %1, %2, ... %8. This was done with the defining word:

```
ARGUMENT <name> ( N -- )
<name> ( -- argN )
```

ARGUMENT creates a new <name> that fetches the argument to the stack.

So much for the theory, let's see some action:

```
: SWAP
  2 ARGS
  ( n1 n2 -- n2 n1 )
  %2 %1  ;

: ROT
  3 ARGS
  ( n1 n2 n3 -- n2 n3 n1 )
  %2 %3 %1  ;

: NIP
  2 ARGS
  ( n1 n2 -- n2 )
  %2  ;

: TUCK
  2 ARGS
  ( n1 n2 -- n2 n1 n2 )
  %2 %1 %2  ;
```

Figure One shows a definition written in three different ways, for comparison. ARGS need not be called first in a definition. It can be called in the middle, and the arguments it provides are usable until the end of the definition. It is not advisable, however, to use ARGS within a loop; it is, in fact, impossible to use in a DO loop.

Normally the arguments are accessed as constants, but it is possible to access them like variables. This is done with the word:

```
ARGV   ( N -- 'argN )
```

that returns the address of argument N. (The name was borrowed from C.)

Definition of local variables is done with the word:

```
LOCALS   ( N -- )
```

that allocates space for N variables. This is identical to ARGS, except variables are uninitialized (and guaranteed to contain garbage, not zeros).

The general word to access a local variable N is:

```
LV   ( N -- A )
```

that pushes the address of the local variable number N to the stack. (If you declare N LOCALS, you will have local variables numbered one through N, like with ARGS.)

For fast access, I have predefined the words L1 ... L8. Also, there are words @1 ... @8 to fetch and !1 ... !8 to store local variables. Their syntax is borrowed from [Bow82]. They are created by special defining words (Figure Two) that are only used to create named local variables. However, this is not recommended—unless you have a transient, independently forgettable vocabulary—because variable names so defined remain accessible to other definitions, as well.

In the accompanying screens are three different implementations of the Tower of Hanoi puzzle. The first implementation uses only LOCALS and ARGS, the second is a mixture of ARGS and conventional code, and the third is just conventional Forth. RECURSE calls the definition recursively. This is in the Forth-83 controlled reference word set, and is called MYSELF in some older Forths. Personally, I prefer using the F83 word RECURSIVE and use the word name itself instead of RECURSIVE (I hope the new standard will include this in the controlled reference word set).

Implementation

Local variables and arguments are implemented by using return stack frames. This means the return stack must be in a range of addressable memory (true of most microcomputer implementations). Additionally, a way is needed to fetch and restore the return stack pointer. In F83, this is done via:

```
RP@   ( -- A )
RP!   ( A -- )
```

that fetch the return stack pointer to the parameter stack, and store the top parameter stack item in the return stack pointer, respectively. (Note that fig-FORTH's RP! behaves differently.)

This implementation presumes that the return stack grows "upwards" (toward lower addresses), that the return stack pointer points to the topmost item, and that a stack item consists of 16 bits. This is true in the majority of implementations. In 32-bit systems, just replace every 2* with 4 * or by CELL* or whatever your system will understand.

LOCALS and ARGS each has its own stack frame (Figures Two and Three). The variables 'LFRAME and 'AFRAME point to the start of the current frame, and data is accessed through them. When a stack frame is built, the previous value of the frame pointer is pushed first, then comes the space for the data and, finally, the previous return stack pointer.

Both LOCALS and ARGS use a coroutine technique to clean up the return stack. On entry, they pop their return address from the return stack and, when the frame is built, perform a call to this address. This means the rest of the definition gets executed (i.e., where ARGS or LOCALS is called). After that, control is returned to ARGS or LOCALS just after the CALL. Then the stack frame is removed and control is returned to the word two levels up. Figure Four shows the execution and calling order of the words FOO and BAR.

Comments

The design goals were met, pretty much, except I couldn't fit the implementation into two screens. Also, there is no clean way to use double-length LOCALS or ARGS, which must be handled in two parts.

To be useful in the real world, the implementation of ARGS and LOCALS should be something like:

```
: ARGS
  (APUSH) CALL (APOP) ;

: LOCALS
  (LPUSH) CALL (LPOP) ;
```

where (APUSH), (APOP), (LPUSH), and (LPOP) should be in code, as should the words LVARIABLE, @LVARIABLE, and !LVARIABLE. And ARGUMENT should be defined as:

(Text, continued on page 41.)

```
: CONVENIENT
  4 ARGS ( x a b c -- a*x*x + b*x + c )
  %2 %1 *  %1 *  %3 %1 *  + %4 +  ;

: THEORETIC
  4 ARGS ( x a b c -- a*x*x + b*x + c )
  2 % 1 % *  1 % *  3 % 1 % *  + 4 % +  ;

1 ARGUMENT x
2 ARGUMENT a
3 ARGUMENT b
4 ARGUMENT c
: ACADEMIC
  4 ARGS ( x a b c -- a*x*x + b*x + c )
  a x * x *  b x *  + c +  ;
```

Figure One. One definition three ways.

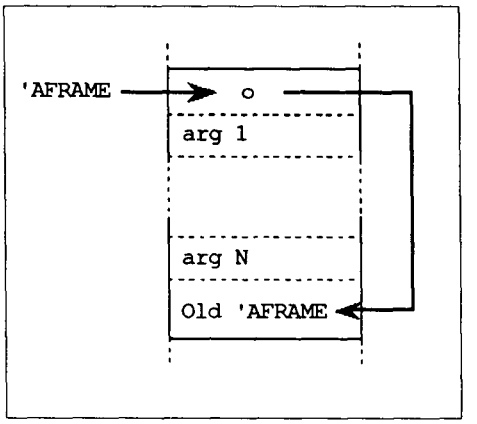


Figure Two. The argument stack frame.

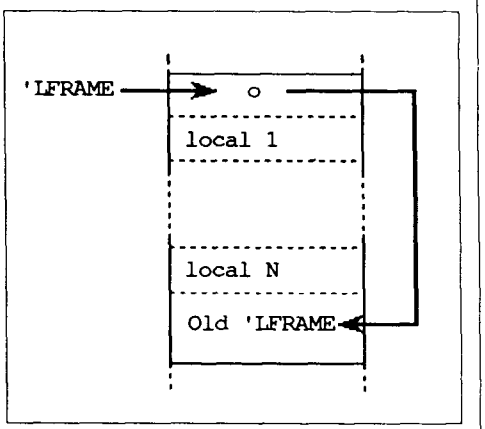


Figure Three. The local variable stack frame.

```
: FOO BAR . ;  ( n1 n2 -- )
: BAR 2 ARGS %1 %2 + ;  ( n1 n2 -- n3 )

FOO ──► BAR ──► 2
                ARGS ──► R>
                         (build frame)
                %1 ◄──── CALL
                %2
                +
                EXIT ──► (remove frame
             . ◄──────── EXIT
◄──── EXIT
```

Figure Four. The execution order of ARGS.

```
     1
 0 \ Local variables and arguments load block          jty 220688
 1 2 3 THRU
 2
 3 VOCABULARY NOVICE  VOCABULARY MIXED
 4 NOVICE DEFINITIONS  4 LOAD
 5 MIXED DEFINITIONS  5 LOAD
 6 FORTH DEFINITIONS  6 LOAD
 7
 8
 9
10
11
12
13
14
15
```

```
     2
 0 \ Local variables and procedural arguments, defining  JTY 190688
 1 VARIABLE 'LFRAME
 2 VARIABLE 'AFRAME
 3 : CALL  ( A -- )  >R ;
 4
 5 : LOCALS  ( N -- )  R>   'LFRAME @ >R   RP@   ROT 2* -   RP@
 6   SWAP RP!   >R   RP@ 'LFRAME !   CALL  R> RP!   R> 'LFRAME ! ;
 7
 8 : ARGS  ( k1..kN N -- )  R>  'AFRAME @ >R
 9   RP@ 'AFRAME !   SWAP                                  BEGIN
10     ROT >R                                      1- ?DUP 0=  UNTIL
11   'AFRAME @ >R   RP@ 'AFRAME !   CALL   R> RP!   R> 'AFRAME ! ;
12
13 : LV    ( N -- 'Ln )  2*  'LFRAME @ + ;
14 : ARGV  ( N -- 'An )  2*  'AFRAME @ + ;
15 : %  ( N -- An )  ARGV @ ;
```

```
     3
 0 \ Local variables and procedural arguments, end user  JTY 190688
 1 : LVARIABLE   ( N -- )  CREATE  C,  DOES>  ( -- 'n )  C@ LV ;
 2 : @LVARIABLE  ( N -- )  CREATE  C,  DOES>  ( -- Ln )  C@ LV @ ;
 3 : !LVARIABLE  ( N -- )  CREATE  C,  DOES>  ( x -- )   C@ LV ! ;
 4 : ARGUMENT    ( N -- )  CREATE  C,  DOES>  ( -- An )  C@ % ;
 5
 6 1  LVARIABLE L1 2  LVARIABLE L2 3  LVARIABLE L3 4  LVARIABLE L4
 7 5  LVARIABLE L5 6  LVARIABLE L6 7  LVARIABLE L7 8  LVARIABLE L8
 8 1 @LVARIABLE @1 2 @LVARIABLE @2 3 @LVARIABLE @3 4 @LVARIABLE @4
 9 5 @LVARIABLE @5 6 @LVARIABLE @6 7 @LVARIABLE @7 8 @LVARIABLE @8
10 1 !LVARIABLE !1 2 !LVARIABLE !2 3 !LVARIABLE !3 4 !LVARIABLE !4
11 5 !LVARIABLE !5 6 !LVARIABLE !6 7 !LVARIABLE !7 8 !LVARIABLE !8
12 1   ARGUMENT %1 2   ARGUMENT %2 3   ARGUMENT %3 4   ARGUMENT %4
13 5   ARGUMENT %5 6   ARGUMENT %6 7   ARGUMENT %7 8   ARGUMENT %8
14
15
```

```
     8
Local variables and arguments load block          jty 220688
The source code is in blocks 2 and 3

There are three versions of the Tower of Hanoi puzzle,
NOVICE that uses only ARGS and LOCALS,
MIXED that uses a mixture of stack manipulation and ARGS and
normal forth version that does not use anything special.
```

```
     9
Local variables and arguments are return stack frames JTY 190688
'LFRAME contains pointer to the current local variable frame.
'AFRAME contains pointer to the current argument frame.
CALL is much like EXECUTE, but takes IP-value instead of CFA.
LOCALS allocates space for N local vars from the return stack
 and sets 'LFRAME to point to the beginning of this frame.
 Previous 'LFRAME is saved to enable recursion. After this it
 issues a coroutine call to the caller. When that returns,
 the return stack is reset to the original value and 'LFRAME is
 restored to previous value.
ARGS behaves like LOCALS, except that it moves N items to return
 stack and set pointer to frame to 'AFRAME .
LV returns the address of local variable number N.
ARGV returns the address of argument number N. The topmost
 argument is number N (like in a stack diagram).
% returns the value of argument number N. See ARGV .
```

```
     10
Local variables & arguments defining & end user words JTY 190688
LVARIABLE creates a word that brings the address of Nth
 local variable to the stack at runtime.
@LVARIABLE creates a word that brings the value of Nth
  variable to the stack at runtime.
!LVARIABLE creates a word that stores the item in stack
 to the Nth local variable at runtime.
ARGUMENT creates a word that brings the value of Nth argument
 to the stack at runtime.

L1 .. L8 are local variables 1 .. 8
@1 .. @8 fetches the values of corresponding variables
!1 .. !8 stores the value on stack to the corresponding variable
%1 .. %8 bring the corresponding argument to the stack
```

```
     4
 0 \ Testing args and local variables: Tower of Hanoi #1 jty 180688
 1 : THIRD  ( t1 t2 -- t3 )  2 ARGS   6 %2 -   %1 - ;
 2
 3 : MOVEDISK  ( from to -- )  2 ARGS
 4   CR ." Move disk from " %1 . ." to " %2 .   ( KEY DROP ) ;
 5
 6 : MOVETOWER  ( from to n -- )  3 ARGS 1 LOCALS   %3 1 =  IF
 7     %1   %2   MOVEDISK                                  ELSE
 8     %1   %2   THIRD   !1
 9     %1   @1   %3 1-   RECURSE
10     %1   %2   MOVEDISK
11     @1   %2   %3 1-   RECURSE                           THEN ;
12
13 : HANOI  ( N -- )  1 ARGS   1   3   %1   MOVETOWER ;
14
15
```

```
     5
 0 \ Testing args and local variables: Tower of Hanoi #2 jty 180688
 1 : THIRD  ( t1 t2 -- t3 )  6 SWAP -   SWAP - ;
 2
 3 : MOVEDISK  ( from to -- )   SWAP
 4   CR ." Move disk from " . ." to " .   ( KEY DROP ) ;
 5
 6 : MOVETOWER  ( from to n -- )  1-                 ?DUP IF
 7     3 ARGS   %1 %2 THIRD
 8     %1 OVER %3 RECURSE   %1 %2 MOVEDISK   %2 %3 RECURSE  ELSE
 9     MOVEDISK                                              THEN ;
10
11 : HANOI  ( N -- )  1 3 ROT MOVETOWER ;
12
13
14
15
```

```
    11
 0 The Towers of Hanoi problem algorithm in ADAish       jty 220688
 1 TYPE tower is (1, 2, 3);
 2 FUNCTION third(a, b: IN tower) RETURN tower IS BEGIN
 3    return 6-a-b;
 4 END third;
 5 PROCEDURE movedisk(a, b: tower); -- Move disk from a to b
 6 PROCEDURE movetower(a, b: tower; n: Integer) IS t: tower; BEGIN
 7    IF (n = 1) THEN -- Is there only one disk to move?
 8       movedisk(a, b);
 9    ELSE -- More than one
10       t := third(a, b);
11       movetower(a, t, n-1);
12       movedisk(a, b);
13       movetower(t, b, n-1);
14    ENDIF
15 END movetower;
```

```
     6
 0 \ Testing args and local variables: Tower of Hanoi #3 jty220688
 1 : THIRD  ( t1 t2 -- t3 )  6 SWAP -   SWAP - ;
 2
 3 : MOVEDISK  ( from to -- )   SWAP
 4   CR ." Move disk from " . ." to " .   ( KEY DROP ) ;
 5
 6 : MOVETOWER  ( from to n -- )  1-                 ?DUP IF
 7     >R   2DUP THIRD
 8     2 PICK OVER R@ RECURSE
 9     2 PICK 2 PICK MOVEDISK
10     SWAP R> RECURSE DROP                                  ELSE
11     MOVEDISK                                              THEN ;
12
13 : HANOI  ( N -- )  1 3 ROT MOVETOWER ;
14
15
```

F83

LOCAL VARIABLES
ANOTHER TECHNIQUE

JOHN R. HAYES - LAUREL, MARYLAND

Another method for adding named local variables is presented here. The method has an aesthetically pleasing syntax and allows the declaration of local variables anywhere within a colon definition, and with the variable being initialized with the value on top of the stack at run time. An efficient implementation is given.

Introduction

The efficiency and clarity of Forth code often suffers from excessive use of stack manipulation words. These words—such as DUP, SWAP, and ROT—are primarily used for operand positioning and, as such, are pure overhead. More importantly, sequences such as SWAP DROP ROT obscure the nature of what the code is really doing. Named local variables would help immensely.

Several ways of creating local variables have been proposed in the past (see references). Inspired by the second reference listed, I have come up with yet another implementation of local variables. My method allows named local variables to be declared anywhere within a colon definition. These declarations can be interspersed with ordinary Forth code. Space for the local variable is dynamically allocated, so code using these variables will be reentrant. In the following paragraphs, I will describe how local variables are declared and how they are implemented. Source code is supplied.

Syntax and Semantics

My local variable method is based on scopes. A scope is a section within a Forth word over which a given local variable is defined. Reminiscent of C, a scope is delimited by braces, i.e., { ... }. Any local declared within the braces can be accessed until the right brace closes the scope. In the following code:

```
: fool  ( a b -- a+b )
  {  local b
     local a

     a b +
  }
  ;
```

> ***"The local variable is also a compiling word..."***

the locals a and b act as parameters for the word fool and using the name a or b causes the local's value to be returned. At run time, a local is initialized with the value on top of the parameter stack. After initialization, the parameter stack is popped. This means that locals can also be used as temporary variables. For example:

```
: foo2  ( a -- ? )
  {  dup *
     local asquared
     0 local temp

      ...
  }
  ;
```

creates a word with two locals, one initialized with the square of the input argument and the other initialized with zero. Arbitrary Forth code can be used to initialize a local.

The scope delimiters are compiling words like if and then and must pair up in the same way. Scopes may also be nested to an arbitrary depth like if ... then pairs. Here is an example of how this might be used:

```
: foo3  ( x y -- ? )
  {  local y
     local x

     x y + x *
     1000 > if
     { 0 local temp1
         x y *
         local temp2

         ...
     }
     then
  }
  ;
```

The words temp1 and temp2 are only allocated if the if test is true. Notice that within the inner scope, the local variables in surrounding scopes can be accessed. If a local variable in an inner scope is declared with the same name as a local in some surrounding scope, the inner definition hides the outer one until the inner scope is closed. This is the same rule used by Algol, Pascal, and C.

Local variables behave like constants in that they return their values. This is the desired behavior, since locals will usually be used to hold function arguments which are not changed. However, a local can be modified by preceding its name by to. This

causes the value on the top of the stack to be stored in the local variable.

Implementation

The syntax and semantics of local variables described above requires that, while compiling a colon definition, it must be possible to add the definition of a local variable to the dictionary without disturbing the colon compilation. Therefore, a temporary allocation region must be set aside for holding the definitions of the local variables. These transient definitions which are only needed at compile time are forgotten and their space reclaimed when their scope is closed. It is also desirable that allocating, initializing, accessing, and deallocating local variables be as fast as possible at run time. Local variables are kept on the return stack. A local is speedily allocated and initialized with the value on the parameter stack by using `>r`. Locals are accessed and deallocated using special code words.

To solve the problem of a temporary allocation region, my Forth system had to be modified slightly. Most Forths have a variable named `dp` that points to the free space at the end of the dictionary. I have added an extra level of indirection (see Figure One) by creating two variables `stdregion` and `regionptr` and by making `dp` a colon definition. The resulting Forth system behaves exactly like the original system. However, since all the dictionary-space management words like `here`, `,` (comma), and `allot` are defined in terms of `dp`, it now becomes possible for a programmer to maintain multiple allocation regions by manipulating `regionptr`.

An allocation region called `locregion` is created to hold temporary dictionary entries for local variables. `local` is both a defining word and a compiling word. When a local variable is declared, `local` switches the current allocation region to `locregion`, uses `create` to make a dictionary header for the local, and records some information that will allow the local's value to be retrieved at run time. `local` then switches the current allocation region back to `stdregion` and compiles a `>r`. The local variable is also a compiling word that compiles code to pick its value from the appropriate location in the return stack. The braces are compiling words that mainly do bookkeeping. The left brace `{` keeps track of how many local variables have been declared. The right brace `}` reclaims the space used by the dictionary entries for the local variables, backs up the current vocabulary pointer, and compiles code to discard the locals at run time.

Figure Two shows what would be compiled on a 32-bit Forth system for the first example given in this paper. The first `>r` creates local variable b and the second `>r` creates a. `(rpick)` and `(rpop)` are code words that take their parameter from the instruction stream. The `(rpick) 0` copies the first 32-bit word on the return stack to the parameter stack, and the `(rpick) 4` copies the second 32-bit word on the return stack to the parameter stack. `(rpick) 8` deallocates both locals by popping two 32-bit words off the return stack.

Allocating local variables on the return stack causes problems for `do` loops. If a `do` loop is used within a scope, it would be nice to be able to access that scope's local variables within the loop:

```
: foo4   ( x -- ? )
  {  local x
     8 0 do
       i x +

       ...
     loop
  }
  ;
```

This is easily accomplished by adding to the definitions of `do` and `loop` a single line of code informing the compiler that two more locals have been declared[1]. It would also be nice to declare a scope within a `do` loop and access the loop index `i` within the scope:

```
: foo5
  8 0 do
  {  10 local x
     x i +
     ...
  }
  loop
  ;
```

```
: fool   ( a b -- a+b )
  {  local b
     local a

       a b +
  }
  ;
```

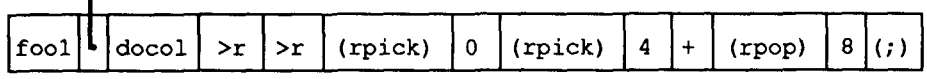


Figure One. Separate allocation regions.

```
variable stdregion     ( standard allocation region )
variable regionptr     ( points to a region pointer )
: dp  regionptr @ ;    ( new definition of dp )
```

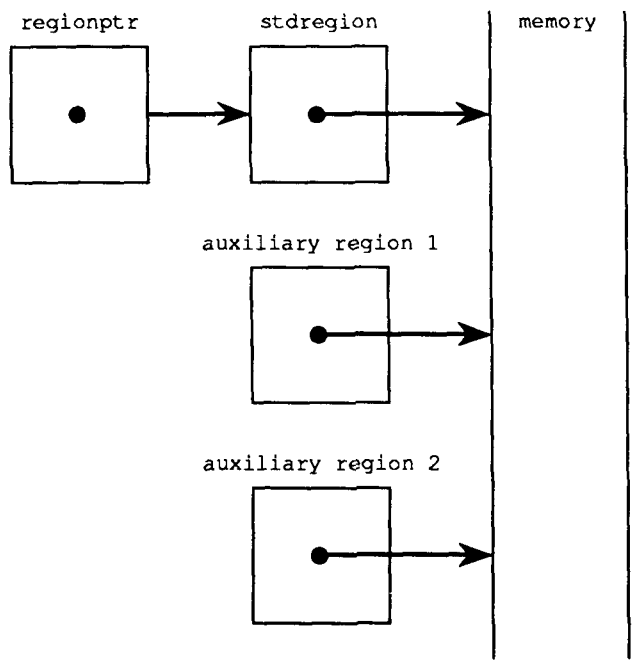


Figure Two. Code compiled for `fool` in example

This situation is harder to accommodate, since a smart `i` would be needed and is therefore not allowed in my implementation. An implementation that uses a third stack for locals would not suffer from this problem.

Summary

Many Forth definitions are simple enough that nothing would be gained by using local variables. However, there are times when a word is forced to juggle many things on the stack. Then the use of local variables both clarifies the code and makes it more efficient by avoiding obscure sequences of stack-thrashing words.

References

Bowhill, S. "Fast Local Variables," *1982 FORML Conference Proceedings*, pp. 142–146.

Glass, H. "The Implementation of Extensions to Provide a More Writable Forth Syntax," 1983 *FORML Conference Proceedings*, pp. 57–68.

Hart, J.R. "Local Variables," *Journal of Forth Application and Research* 3,2 1985, pp. 159–162.

Jekel, R.N. "Local Variables for Forth," 1980 *FORML Conference Proceedings*, pp. 59–63.

Korteweg, S., Nieuwenhuyzen, H. "Stack Usage and Parameter Passing," *Journal of Forth Application and Research* 2,3 1984, pp. 27–50.

La Quey, R.E. "Local Variables," *1984 FORML Conference Proceedings*, pp. 307–316.

Volk, W. "Named Local Variables in Forth," *1984 FORML Conference Proceedings*, pp. 347–357.

John Hayes is the author of several Forth articles and a key figure in the VLSI Forth microprocessor project at the Applied Physics Laboratory of Johns Hopkins University.

1. This assumes that the implementation of loops keeps two items on the return stack.

```
\ The following words used in the code are not standard Forth-83 and
\ may not be familiar to everyone.  They provide a way to write word
\ size independent forth code.
\ al+   [ n --- n+sizeof[word] ]  equivalent to 2+ on 16 bit forths,
\                                 equivalent to 4+ on 32 bit forths
\ al-   [ n --- n-sizeof[word] ]  equivalent to 2- on 16 bit forths,
\                                 equivalent to 4- on 32 bit forths
\ a*    [ n --- n*sizeof[word] ]  equivalent to 2* on 16 bit forths,
\                                 equivalent to 4* on 32 bit forths
\

\ The following code words for accessing locals on the return stack are
\ also used.  All three code words use an inline parameter which follows
\ it in the instruction stream.
\ (rpick)  [ --- x ]    return a return stack item n bytes from the top
\ (rstore) [ x --- ]    store x on return stack n bytes from the top
\ (rpop)   [ --- ]      pop n bytes off the return stack
\ In all the above n is found in the instruction stream.

\ 1. The code in this section must be incorporated into the forth kernel

hex

variable stdregion                      \ standard allocation region
variable regionptr                      \ current dictionary allocation region

variable outerdepth                     \ records total number of temporaries
                                        \ created in surrounding scopes.
variable currentdepth                   \ records number of temporaries made
                                        \ in current scope so far.

\ define dp to use an extra level of indirection

: dp   regionptr @ ;    \ ( --- addr )

\ modifiy do and loop to work with scopes

: do            \ ( --- clue here )
 2 currentdepth +!                      \ allow for local variable scopes
 [compile] do                           \ traditional do
; immediate

: loop          \ ( clue here --- )
 -2 currentdepth +!                     \ allow for local variable scopes
 [compile] loop                         \ traditional loop
; immediate

: +loop         \ ( clue here --- )
 -2 currentdepth +!                     \ allow for local variable scopes
 [compile] +loop                        \ traditional +loop
; immediate

\ 2. The code in this section can be incorporated in the forth kernel or
\ loaded into the forth system when needed.

\ Block structure extensions

: region        \ ( size --- ) define allocation region of given size.
 create here al+ , allot
 does> ;        \ ( --- regionptr )
```

```
: allocatefrom  \ ( regionptr --- ) start allocating space from given region.
 regionptr ! ;

200 region locregion                        \ create region for making temporary
                                            \ dictionary entries for locals.

: (             \ ( --- oldcurrentdepth region ) begin a scope.
 currentdepth @ dup outerdepth +!
 0 currentdepth !
 locregion @ ; immediate                    \ this allows space to be recovered

: )             \ ( oldcurrentdepth region --- ) recover space used by
                \ local headers, unwind vocabulary, compile code to clean
                \ up return stack, and restore depth variables.
 dup locregion @ =
 if drop                                    \ if no locals used, do nothing
 else                                       \ otherwise,
    dup name> >link @ current @ !           \    back vocabulary pointer
    locregion !                             \    deallocate space
 then
 currentdepth @ ?dup if
    compile (rpop) a* ,                     \ compile code to clean up rstack
 then
 dup currentdepth !                         \ restore depth counters
 negate outerdepth +! ; immediate

variable to?                                \ 0=@, 1=!
: to    1 to? ! ; immediate

: local         \ ( --- ) create a local variable dynamically allocate
                \ its header from the temporary region.  at run time,
                \ dynamically allocate space for the local on the return
                \ stack.
 locregion allocatefrom                     \ create dict. entries in temp region
 create
    outerdepth @ currentdepth @ + ,         \ record offset from bottom
    1 currentdepth +!
    immediate
 stdregion allocatefrom                     \ revert to standard allocation region)
    compile >r                              \ initialize local
 does>          \ ( addr[offset] --- ) offset is in words from the bottom.
    to? @ if                                \ if to local
       compile (rstore)  0 to? !            \ copy local from pstack to rstack
    else
       compile (rpick)                      \ copy local from rstack to pstack
    then
    @ outerdepth @ currentdepth @ + swap - 1- a* , ; immediate
```

PREFIX FRAME OPERATORS

JOSE BETANCOURT - SUNNYSIDE, NEW YORK

While contemporary operating systems and user environments require many parameters as part of function calls and procedures, Forth is optimized for only three stack items—and manipulating even these few can be aggravating. For example, consider that a simple task such as:

```
( a b c -- a-b a-c )
```

"Lazy variables and stack extension increase its usefulness."

will require many stack operations. To eliminate this problem, we can utilize local variables via a prefix frame operator design that allows for utility, readability, and compactness.

This method was developed on a 68000-based, 32-bit, call-threading Forth system. Except for internal structure, memory addressing, and system dependencies, this system adheres to the Forth-83 Standard. It has a few differences that are straightforward: `@` is a 32-bit fetch, `W@` is a 16-bit fetch, and `C@` is an eight-bit fetch. The comma and store words are similar.

Implementation

A stack frame is space allocated in a stack for storage of local variables and parameters. Using a stack frame allows reentrant subroutines or procedures. Frames are so useful that some general-purpose microprocessors have built-in structures to create them. For example, the MC68000 has the `LINK` and `UNLK` instructions.

```
: SORT  ( cfa.array cfa.compare cfa.exchange  start end )
   L( ARRAY COMPARE EXCHANGE START END \ SWITCH N1 N2 )
   BEGIN   FALSE IS SWITCH    L END  L START
     DO  I  GO ARRAY  @ IS N1  I 1+ GO ARRAY IS N2
         L N1 L N2  GO COMPARE
         IF  TRUE IS SWITCH
             I I 1+ L N1 L N2 L ARRAY   GO EXCHANGE
         THEN
     LOOP   L SWITCH FALSE =
   UNTIL    S(   ) ;

( Use:  ' DATA ' > ' EXCHANGE 1 6 SORT  will do an ascending sort )
```

Figure One. Bubble sort.

```
: SOLVE   \ ( a b c -- [b-(b^2-c)]/2a  [b+(b^2-c)]/2a  true ¦ false )
        ROT DUP Ø <>   \ denominator not zero?
        IF  2* L( b c 2a \ p ) L b DUP * L c - IS p
            SUB b p L  2a /    SUM  b p  L 2a /   TRUE
            S(  difference sum true )
        ELSE   3 NDROP   FALSE   THEN   ;     \ ( 1 2 4 gives 1 1 -1 )
```

Figure Two. Formula computation.

```
: SUMMING
   1 2 3 4   L(  # 4  \ RESULT )
   L M L N L P L Q  + + +  IS RESULT     AT RESULT  ?   S(   ) ;

   ( 1Ø  should be printed on screen
```

Figure Three. Using lazy variables.

```
: $=     ( $1 $2 --- flag )     L ( S1 S2 \ FLAG LEN )
    L S1 C@ 1+   IS LEN     TRUE IS FLAG   SUM  LEN S1  L S1
    DO  L S2 C@ I C@ <>  IF  FALSE IS FLAG   LEAVE  THEN  ++ S2  LOOP
    L  FLAG    S( # 1 ) ;    \  true== equal
```

Figure Four. String word rewritten.

```
CODE DISK.FREE          ( 16byte.buffer.address  )  \ call GEMDOS #36
   R -)  Ø # MOVE .W  R -) S )+ MOVE .L
   R -)  $36 # MOVE .W  1 TRAP   R AR 8 ADDQ .L   NEXT

: ?DISK.BYTES.FREE     ( ---bytes )
   L(   \ SECTOR/ALU  SECTOR.SIZE  TAU  FAU )     AT FAU DISK.FREE
   L FAU L SECTOR/ALU *  L SECTOR.SIZE *   S(  BYTES )  ;

COMMENT:  DISK.FREE expects the address of a buffer where it will put
  disk information in the order:  fau tau sector.size sector/alu.
  ?DISK.BYTES.FREE gives it the address of a frame created for that
  purpose, and then uses the data to compute bytes free.
COMMENT;
```

Figure Five. Using frame as buffer.

Surprisingly, the Forth virtual machine does not have a standard method of creating frames. Therefore, primitives must be written that do so. The method used here requires only one variable, which should be a host register for speed, the return stack for re-entrancy, and the existing parameter. Using the Forth stack saves time, whereas, in some local variable proposals, an initial relocation of items to another array or local stack does not. To identify labels, a few simple string operators are used. This makes it more portable to different Forth systems and saves dictionary space.

Method

A frame for use within a single definition is created with `L (`. It parses the input stream until the closing) and creates a string array above HERE. This array is then pointed to by LPTR. The count of blank-delimited strings is the number of local variables needed. For example, the phrase:

```
L( length width height )
```

will create a frame with three items labelled "length," "width," and "height."

Two more capabilities were added to increase usefulness: lazy variables and stack extension. Lazy variables are created when the character # is followed by a number in the string array. Lazy variables are one-character strings comprised of the letters M, N, P, and so on. (To avoid confusion with the number zero (0), the letter O is not used.) They are useful when one needs to quickly write a procedure and descriptive labels are not important, or when one *is* lazy and prefers to use one-letter labels.

In Figure Three, lazy variables are used in the word SUMMING. Four numbers on the stack are assigned to four local variables named M, N, P, and Q via the phrase `# 4`. These are then used to compute a simple sum.

The other capability can be used to extend the size of the frame. When local variables are allocated, they are based on the count of items in the string at `LPTR @`. This count cannot exceed the number of parameters which will be on the stack at the definition's run time, since a frame is made *within* the stack, not by extending the stack. But if we need three local variables, for example, and there are only two items on the stack, there will be no room for the frame. One creates the necessary room by preceding the names with a \ (backslash). This will cause the compiler `L (` to execute EXTSTK, which compiles in-line, stack-extension code.

In Figure Three, SUMMING uses the backslash option. The four lazy variables are used to compute the sum, but another variable is needed to store it. This is done with the phrase:

```
\ RESULT )
```

The primitive `S (` is much simpler. It parses the input stream as above, but it compiles (SOME) or (NONE) depending on the number of parameters returned at run time. These parameters can be named explicitly or with the character # followed by the count. The "*# count*" alternative is used when the stack contents are obvious. For example, if the last words in a phrase are `L FLAG` (leave contents of FLAG on the stack), one can write `S ( # 1 )` to avoid repeating the word 'flag' in the parameter string. Of course, in this example a more descriptive name could be used instead, such as:

```
S( key_found? )
```

The prefix operators L, IS, and AT access frame items by parsing the in-line string in a definition and searching for a match in the string array at LPTR, then compiling the associated run-time prefix and offset. These prefixes are the minimum necessary to provide full use of prefix frame operators. However, to accommodate application needs as well as programming style, user-defined prefix operators can be created. To provide vectored execution, for example, a prefix called GO was

```
\ Simple string words

: $=      ( $1 $2 --- flag )     \ compare strings, same=true flag
       -1 -ROT  DUP C@  1+   OVER + SWAP
       DO   COUNT I C@ <>   IF SWAP DROP Ø SWAP LEAVE THEN
       LOOP   DROP ;

: GETNAMES   \ name1 .... nameN )  ( adr --- count ) count is #items
      \ get instream labels and form string array at adr
      \ form:  [ 1len |$....|1len |$.........|1len|$...| Ø | ]
      Ø SWAP BEGIN 32 WORD DUP 1+   C@ ASCII )  <>
             WHILE ROT 1+  -ROT  2DUP C@ 1+ >R  R@ CMOVE R> +
            REPEAT  DROP     Ø SWAP C!   ;

: $SRCH    ( $adr  $array --- offset flag )  \ true = found
       Ø Ø 2SWAP   ( count flag $adr bufadr )
       BEGIN  DUP C@  Ø=  3 PICK  OR NOT \ not end or not true flag
          WHILE   2DUP $= IF  ROT DROP  -1 -ROT  \ change flag to true
                  ELSE   >R ROT 1+ -ROT  R>   COUNT  + THEN
       REPEAT    2DROP    ;

VARIABLE  LPTR    \ name buffer pointer for temporary string array

: SRCH.CHAR  ( adr char -- adr )  \ search for char starting at adr
    BEGIN  SWAP COUNT   SWAP -ROT  OVER  =   UNTIL  DROP 1-  ;

: REMOVE\      \ remove "\" from name list at $array
    LPTR @ DUP ASCII \ SRCH.CHAR  SWAP Ø SRCH.CHAR  ( a\ aØ )
    SWAP DUP >R  1+  SWAP  R@ - R>  1-  SWAP CMOVE ;



\  Lazy variables

 : >SHOVE    ( $buffer n )  \ make room for lazy names in $array
    SWAP >R 2*  R@ + R>   4+ SWAP OVER \  ( 4+a  2n+a  4+a )
    LPTR @ Ø SRCH.CHAR SWAP - 1+    CMOVE>   ;

: SKIP.O   ( c---c' )  \ if c is "O" convert to "P" using hybrid logic
    DUP ASCII O =  1 AND + ;

: RESERVE    ( $buffer n )    \ create lazy variables  M,N,P,.....etc.
    2DUP >SHOVE   2*   2DUP  1 FILL
    ASCII  M -ROT  SWAP  1+ SWAP
    OVER + SWAP
    DO  SKIP.O DUP I C!  1+  2 +LOOP   DROP ;

: ?LAZY    ( #parsed ---new.number )    \ make lazy vars if neccessary
    LPTR @ W@  291 =     \ first name is "#" ?
    IF  Ø LPTR @ 2+  CONVERT  2DROP (  n # )
        LPTR @ OVER  RESERVE   + 2-
    THEN   ;
```

defined in block 83. It is used in Figure One to help define a generic bubble sort. The word SORT expects code field addresses on the stack that specify what kind of data structure is being sorted: strings, numbers, complex numbers, etc.

In addition to GO, a few more prefixes are shown, such as the "implicit" SUM. SUM replaces the phrase `L one L two +` with the phrase `SUM one two`. These implicit prefixes make the source code more compact and readable, as illustrated in Figure Two.

(FRAME) is the run-time frame creator; it saves the present contents of FPTR on the return stack and then uses the in-line number to add to the stack pointer, which will now point to the bottom of the frame. (SOME) and (NONE) are the run-time unframing words which unstack the FPTR and save or lose the items in the frame by resetting the stack pointer.

Examples

The structure compiled by prefix frame operators is seen by this simple definition:

```
: TEST  ( a b -- total )
  L( A B \ TOT )
  SUM A B IS TOT
  L TOT
  S( # 1 )  ;
```

This compiles as:

```
EXTSTK 1 JSR (FRAME) 3 JSR (SUM) 0 1 JSR (IS) 3 JSR (L) 3 JSR (SOME) 1 RETURN
```

(*Note:* for simplicity, the actual compiled offsets and code are not shown.)

An unexpected benefit of prefix frame operators is the ability to use the frame as an array, as illustrated in Figure Five, where the word ?DISK.BYTES.FREE is defined. This word is used to determine the amount of free disk space.

The required data is provided by DISK.FREE which expects an address on the stack of a sixteen-byte buffer where it will store disk parameters in this order: FAU, TAU, SECTOR.SIZE, and SECTOR/ALU. ?DISK.BYTES.FREE creates this buffer as a frame with a name for each of these parameters. Note that the names in this list are in reverse order. This is because, in this Forth system, the parameter stack grows toward lower memory. As a consequence, so does the frame.

```
\  Stack extension for uninitialized local variables

CREATE SLASH   1 C,  ASCII \ C,     \   the letter "\" as a string

: EXTSTK   ( n )   \ compile inline stack extension code
    [ ASSEMBLER ]  S SWAP  # SUBA   W  [  Ø 138 U+ ! ]  ;

: ?TEMPS     ( #parsed --new# )  \ stack extension if needed
    SLASH LPTR @  $SRCH   IF  OVER SWAP - 4* EXTSTK   REMOVE\
                          ELSE   DROP  THEN   ;

\  Framing

VARIABLE  FPTR      \  frame pointer.  Should be register for speed.

: (FRAME)           \ run-time formation of frame WITHIN the stack!
    \ old fptr is put under return address on return stack
    'S  R> DUP  W@ SWAP 2+          \ 's n return )
    FPTR @ >R  >R  + FPTR !  ;      \ (  )  (R   oldfptr  return )
                                    \ fptr=sp+4n

: L(    \ name..)¦name..\ temp..)¦\ temp ..)¦# n)¦# n \ temp..)
    ?COMP  HERE 1Ø24  + DUP LPTR !  GETNAMES  DUP
    Ø=   ABORT"  Empty parameter list!"  ?LAZY   ?TEMP
    1-  Ø MAX  4*   COMPILE (FRAME)   W,  ;  IMMEDIATE


\ Unframing

CODE SP!    ( adr )   S AR  S )+  MOVE  .L  NEXT  \  load stack pointer

: (NONE)        \ run-time stack reset leaving no parameters
   R> R> FPTR @ SWAP FPTR ! SWAP >R 4+  SP!  ;

: (SOME)        \ run-time stack reset leaving parameters
    'S  R> DUP W@  SWAP 2+  R> SWAP >R   FPTR @ SWAP FPTR !
    SWAP DUP >R  -  4+  SWAP OVER R>   CMOVE>   SP!   ;

: S(    \ names...)¦# n names..)¦# n )      compile frame release
    ?COMP   LPTR @ GETNAMES  DUP
    IF  LPTR @ W@  291 =   \  291 is string "#"
        IF  DROP Ø LPTR @ 2+   CONVERT 2DROP
        THEN  COMPILE (SOME)   4*  W,
    ELSE  DROP  COMPILE (NONE)    THEN  ;    IMMEDIATE

\ Prefixing

: ?LOCAL      \  name (   )  compile name's frame offset
    32 WORD  LPTR @ $SRCH   Ø=  IF  HERE COUNT TYPE
    ABORT"  is local?"  THEN  4*  W,  ;

: 'N   \  ( --adr )  get adr of stack cell using inline number
    FPTR @  R> R>  DUP W@ SWAP 2+  >R SWAP >R  - ;


\  prefixes.  These are simular to QUAN or VALUE variables
\  run-time code                                    ( C terminology )
: (L)    ( --parameter )  'N @ ; \ fetch           ( rvalue )
: (IS)   ( n --- )     'N ! ;    \ store           ( :=     )
: (AT)   ( ---adr )    'N   ;    \ address         ( lvalue )

\ compile time code
: L       \ name  ( --n ) call to lvar fetch
    ?COMP   COMPILE (L)  ?LOCAL  ;  IMMEDIATE
: IS      \ name ( n -- )  call to store
    ?COMP  COMPILE (IS)   ?LOCAL  ; IMMEDIATE
: AT      \ name ( --- adr )  call pointer
    ?COMP  COMPILE (AT)   ?LOCAL  ;  IMMEDIATE

COMMENT:   The prefix defining words should be made into a compact
    create...does>   structure.  Was not done here since the Forth sys
    used does automatic inline code expansion, and I could not tame it.
COMMENT;
```

```
\ Extensions, Byte and increment operators.

: (GO)     'N @ EXECUTE ;          \ vectored execution

: (LC@)    'N @ C@  ;              \ fetch character
: (LC!)    'N @ C!  ;              \ store character
: (++)     'N 1+!   ;              \ increment by 1
: (--)     'N 1-!   ;

: GO             \ vectored execution of CFA in local variable.
                 \ Compile:  name  ( --- )
    ?COMP  COMPILE  (GO)   ?LOCAL  ;  IMMEDIATE
: LC@    ?COMP  COMPILE (LC@)   ?LOCAL ; IMMEDIATE
: LC!    ?COMP  COMPILE (LC!)   ?LOCAL ; IMMEDIATE
: ++     ?COMP  COMPILE (++)    ?LOCAL ; IMMEDIATE
: --     ?COMP  COMPILE (--)    ?LOCAL ; IMMEDIATE

\ Implicit extensions.  Dual operators.

: 'NN    ( ---A1 A2 )  \ adr of next two parameters using inline numbers.
        FPTR @ R> R>  DUP 4+ >R        \ ( f 1ret 2ndret )  (R  ret )
        SWAP  >R DUP W@ SWAP 2+ W@     \ ( f n1 n2 )   (R  ret 1ret )
        >R  OVER  R> -  >R - R>  ;     \ ( a1 a2 )  (R  ret 1ret )

: (SUM)     'NN  @ SWAP @ + ;             \  implicit addition
: (SUB)     'NN @  SWAP @ SWAP - ;        \  implicit subtraction
: (MOVEC)   'NN  SWAP @ C@ SWAP @ C! ;    \  implicit byte copy


: SUM            \ ( -- sum )  Compile:  name1 name2
  ?COMP COMPILE (SUM)  ?LOCAL ?LOCAL  ; IMMEDIATE
: SUB            \ ( --difference )  Compile:  name1  name2
  ?COMP COMPILE (SUB)  ?LOCAL  ?LOCAL ;  IMMEDIATE
: MOVEC          \ (  )   Compile:  name1 name2
  ?COMP   COMPILE  (MOVEC)  ?LOCAL  ?LOCAL  ; IMMEDIATE


\ run-time words coded in 68ØØØ machine language using Forth assembler.

: M.L    [ ASSEMBLER ] MOVE .L ;     \ abbreviation

CODE (FRAME)
   Ø AR S AR M.L        1 AR R )+ M.L     R -) FPTR @#L M.L
   Ø DR 1 )+ MOVE .W    R -) 1 AR M.L        Ø Ø DR ADDA .W
   FPTR @#L  Ø AR M.L   NEXT
CODE (NONE)
   Ø DR FPTR @#L M.L    FPTR @#L 4 R I) M.L
        R ) R )+ M.L         Ø DR  4 ADDQ .L
      S AR Ø DR M.L     NEXT
CODE (SOME)
       Ø DR S AR M.L          Ø AR R )+ M.L       1 DR FPTR @#L M.L
   FPTR @#L R )+ M.L       2 DR Ø )+ MOVE .W      2 EXTL
    1 DR 2 DR SUB .L          1 DR 4 ADDQ .L          R -) Ø AR M.L
       S -) Ø DR M.L           S -) 1 DR M.L          S -)  2 DR M.L
    ' CMOVE>  @#L JSR          S AR  1 DR M.L      NEXT
CODE 'N
   Ø DR FPTR @#L M.L      1 AR 4 R I)  M.L
   1 DR 1 )+ MOVE .W      1 EXTL       4 R I) 1 AR M.L
    Ø DR 1 DR SUB .L      S -) Ø DR M.L     NEXT
CODE 'NN
   Ø DR FPTR @#L M.L         1 DR Ø DR M.L     Ø AR 4 R I) M.L
   1 AR Ø )+ MOVE .W      2 AR Ø )+ MOVE .W     4 R I) Ø AR M.L
   Ø DR  1 AR SUB .L      1 DR 2 AR SUB .L        S -) Ø DR M.L
       S -) 1 DR M.L      NEXT
```

Buffer names are, therefore, listed backward. Fetching the address of the variable FAU returns the beginning address of the buffer. This address is passed to DISK.FREE which stores the data; the rest of the definition uses that data to calculate the number of free bytes on the disk.

Problems

One obvious compromise in this design is the use of the prefix L, which may appear awkward and redundant. However, to eliminate its use would have made the implementation more complicated. An alternative is to modify the Forth outer interpreter so that, instead of aborting, a call to an interpretive vector is made. Another alternative is to use a special bodiless vocabulary, where only heads and frame offsets are temporarily stored.

Application

Frame operators are more practical at the beginning of a project to speed up prototyping, changes, and testing. Forth novices can also use local variables to ease the stack burden. Such stack juggling is not required in more popular languages.

Local variables may not be required if definitions are kept short and the stack is kept shallow. Yet, unless there is rigorous testing of Forth code to determine the actual costs of all factors, i.e., nesting level, stack depth, vectoring, etc., will we really know what is most efficient in terms of performance or development? Are small definitions really more productive? Does the name explosion and deep nesting levels reduce the advantage?

Conclusion

Forth can easily be extended to provide local variables; the approach presented here uses prefix frame operators. If the prefixes are viewed as defining the properties of a stack object, this may also constitute a simple, object-oriented approach.

If there is a demand by *users* for local variables, maybe it should become an optional extension to the standard Forth wordset.

Jose Betancourt is a field service technician for Brinkmann Instruments, Inc. In spare minutes, he tinkers with Forth or experiments with hardware using a PC and the single-chip F68HC11.

FORTH NEEDS THREE MORE STACKS

AYMAN ABU-MOSTAFA - SEAL BEACH, CALIFORNIA

Standard Forth uses the return stack for a number of tasks: storing return addresses, storing indexes and limits of DO loops, and for temporary storage. This is bad programming. This article suggests the use of an auxiliary stack for loop parameters and temporary storage, thus reserving the return stack for return addresses only.

Furthermore, the way standard Forth implements conditionals such as IF and ELSE is through branching forward or backward to a given address. This prohibits the use of conditionals outside colon definitions. In this article, I show a simple method of handling conditionals without branching. Indeed, branching as a way to implement conditionals can be totally eliminated. This is done by adding two stacks: a condition stack and a case stack.

The Auxiliary Stack

One stack should be reserved for storage of loop indexes and for temporary storage. This will free the return stack to be used only for return addresses, eliminating a big source of errors that arise when the programmer forgets to use R> after >R, for example. Standard Forth words which will be affected are: >R, R>, I, and J. The first two of these should not be changed; they should continue to work on the return stack. Two additional words, perhaps named >X and X>, may be used to move to and from the auxiliary stack. However, I suggest the more general pair of words >S and S>, which expect two numbers on the stack: the value to be moved and the number of the stack to move it to. Stacks will be numbered zero through four, where stack zero is the parameter stack. Thus, >X may be defined as:

```
: >X  2 >S ;
```

The words I and J will operate on the new auxiliary stack. R@ will not change. One can also define a word S@ which will return the top of any stack without altering it, e.g.:

```
: I   2 S@ ;
: R@  1 S@ ;
```

The addition of this auxiliary stack does not change the structure of Forth.

"The return stack will be used only for return addresses."

The Condition Stack

The Forth-83 Standard specifies a host of primitive words to handle the work of conditionals. These words are: BRANCH, ?BRANCH, MARK>, <MARK, RESOLVE>, and <RESOLVE. All this is in addition to IF, ELSE, and THEN. Furthermore, these three conditionals cannot work in interpretive mode (outside a colon definition) since branching necessitates a jump to an address. This is why the standard suggests three other primitive conditionals for interpretive mode, namely IFTRUE, OTHERWISE, and IFEND. However, these three words do not allow nesting of conditions.

There is a rather simple way to handle conditionals both inside and outside colon definitions, including nesting, and to eliminate the need for the six branching primitives. This can be done by implementing a condition stack. Here is how:

1. As the interpreter parses the input buffer, it also checks the top of the condition stack. If that flag reads true, the interpreter proceeds to interpret the current word as usual. If the top of the condition stack reads false or don't_care, the interpreter ignores the word and goes on to parse the next word.
2. Some words will have to be honored regardless of the condition stack. Among these "must exec" words are those that change the condition stack, namely IF, ELSE, and THEN. They will be marked by setting a bit in their name field address—I suggest the second most significant bit.
3. The condition stack is initialized by pushing the true flag on it at startup. The system will check that at least this value is on the condition stack. In other words, the condition stack will never be empty.
4. The word IF will move the flag from the parameter stack to the condition stack *if the flag on top of the condition stack is true*; otherwise, it will push don't_care onto the condition stack and drop the flag. This way, conditionals can be nested to as many levels as the maximum depth of the condition stack.
5. The word ELSE will simply reverse the top of the condition stack. (The reverse of don't_care is the same.)
6. The word THEN will simply pop the condition stack.

A possible implementation of the above is as follows (assuming 1 is the true flag, -1 is false, and 0 is don't_care):

```
: REVERSE
  3 S> NEGATE
  3 >S  ;

: IF
  3 S@
  IF 3 >S
  ELSE DROP
    0 3 >S
  THEN  ;

: ELSE
  REVERSE  ;

: THEN
  3 S> DROP  ;
```

These definitions will require a change to INTERPRET and will give significance to the second most significant bit of a word's NFA.

In addition to eliminating branching, this approach makes it easy to use conditionals in interpretation mode without further changes to INTERPRET and without the addition of new primitives. They can be nested to any level, as well.

The Case Stack

The case statement's usefulness is nearly undebatable. However, implementing it in standard Forth is not easy. By using a special stack—the case stack—the case statement implementation becomes very easy.

1. The case statement structure will look like this:

x SELECT
n1 CASE (case 1...) ELSE
n2 CASE (case 2...) ELSE
...
nm CASE (last case...) ELSE
END *m* CASES

2. The word SELECT moves the value to be compared, *x*, from the parameter stack to the case stack. I.e.,

```
: SELECT  ( n -- )
  4 >S  ;
```

3. The word CASE compares the case value with the current value on the parameter stack and passes the resulting flag to IF. I.e.,

```
: CASE  ( n -- )
  4 S@ = IF  ;
```

4. The word ELSE is the same word used with the condition stack.
5. The word END will set up the top of the condition stack so that the interpreter will not ignore the words that follow. The condition stack top will not read TRUE at the end of the case statement unless the last (or default) case is true. END is defined as follows:

```
: END
  4 S> DROP
  1 4 >S  ;
```

6. If there is no default case, the last ELSE will be redundant.
7. Finally, the word CASES will pop off the condition stack as many flags as there were cases, then pop the case stack to end this case statement. I.e.,

```
: CASES
  0 DO  THEN LOOP
  4 S> DROP  ;
```

Again, we can now use THEN independently in a definition.

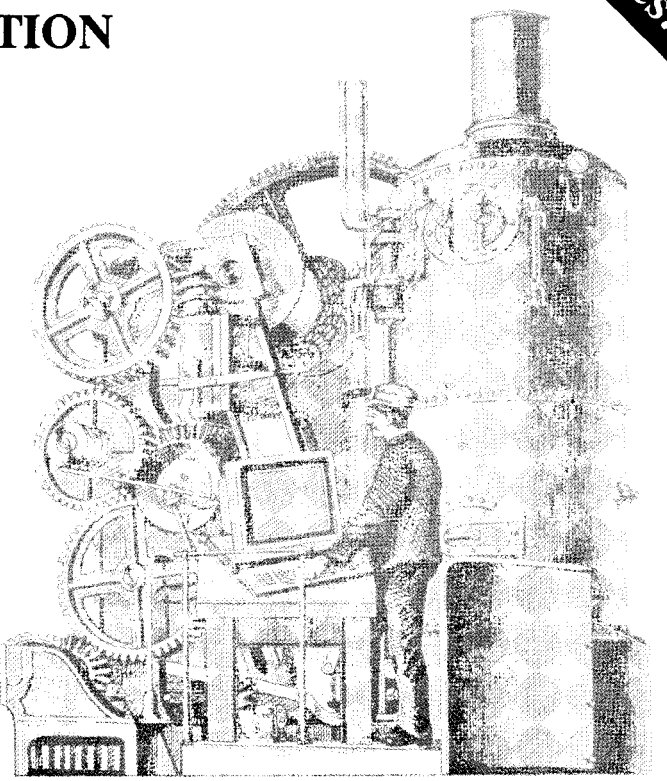
This structure allows the programmer

(Continued on page 41.)

New Dates!

1989 ROCHESTER FORTH CONFERENCE ON INDUSTRIAL AUTOMATION

June 20 – 24, 1989
University of Rochester
Rochester, New York



Over 60 presentations on the state of the art in Forth and threaded interpretive languages including:

Dr. Sergei Baranoff, *"From Russia With Forth"*

Five invited speakers covering:

- Real time control
- Development Systems for Embedded Controllers
- Machine Vision
- Engine Testing
- Smalltalk-like Classes and Objects for Process Control

Managing software and programmers under Forth

ANS X3J14 Forth Standard Working Groups

Tour the Laboratory for Laser Energetics

- See Fusion and Forth

REGISTRATION

The registration fee and conference services includes all sessions, meals, and the Conference papers. Lodging is available at local motels or in the UR dormitories. Registration will be from 4 – 11 PM on Tuesday, June 20th in the Wilson Commons, and from 8 AM Wednesday, June 21st in Hoyt Hall, where sessions will be held.

Note: Due to lodging conflicts caused by the U.S. Open Golf Tournament, the Conference will take place one week later than previously announced.

Name _____

Address _____

_____

_____

Telephone: Wk (____) _____

Hm (____) _____

Registration:

☐ $200.00 ☐ $150.00 (UR Staff and IEEE Computer Society)
_____ IEEE #

☐ $50.00 (full-time students)

☐ Vegetarian Meal Option

Total $ _____

Conference Services $ 200.00

Dormitory housing, 5 nights

☐ $125.00 single ☐ $100.00 double

☐ non-smoking roommate ☐ 5K Fun Run

Total $ _____

Amount Enclosed $ _____

MC/Visa# _____ exp _____

Please make checks payable to the Rochester Forth Conference. Mail your registration to Rochester Forth Conference, Box A, 70 Elmwood Avenue, Rochester, NY 14611, USA.

Forth-83

8250 UART REVISITED

BRIAN FOX - LONDON, ONTARIO, CANADA

About a day before the November/December issue of *Forth Dimensions* (X/4) arrived in the mail, I was involved with interfacing an MS-DOS computer to a broadcast-quality video tape recorder. The VTR was equipped with an RS-422 serial communications interface, so we were using a commercially available RS-232/RS-422 converter. The protocol was developed by SONY specifically for controlling video tape equipment. We had a copy of the complete protocol specification, so we thought we were ready.

The protocol specified communication at 38.4 Kbps, something I had never done with my computer. I use a version of HS/FORTH by Harvard Softworks that gives me the ability to call DOS' MODE command from within the Forth shell, so I had never bothered to really learn all that I should have about the details of the 8250 UART chip. Of course, with the MODE command I could only turn the 8250 up to a rate of 9600 bps, so—with text in hand—an associate (who likes to program in C, but other than that he's normal) and I proceeded to develop a feel for the details of controlling this handy little chip, using Forth as an R&D tool, of course. Later, I put our work together in a lexicon of words to control the communication protocols only, since HS/FORTH contains all the neccessary words for transmitting and receiving bytes.

It was interesting to see the different approach taken by Mr. Paul Cooper in last November's *Forth Dimensions*. His is the approach taken by programmers in more conventional languages. That is, build a complete program designed to do a specific job. The disadvantage of this is that the program is seldom completely reuseable in other applications.

The Forth programmer has the freedom to be quite different. To quote Brodie, "Forth programming consists of extending the root language toward the application, providing new commands that describe the problem at hand." This approach ensures that, in later applications, time and effort will not be wasted solving the same problems over again, since the needed words can readily be incorporated from past efforts. This is particularly true if the problem is factored into small components that have a very specific but simple function.

"Forth is the langauge of choice for hardware interfacing."

This lexicon of words consists of a set of primitives and constants that allow access to the underlying hardware functions. These are then controlled by a set of high-level words that mimic their English-language equivalents. Rather than step through a set of menus to make a selection, the Forth approach allows you then to talk directly to the chip. After loading this lexicon, one can issue the command `1200 BAUD` to the 8250 from a screen of source code or directly from the keyboard, and the chip will understand (in a matter of speaking).

Interface Primitives

I have chosen here to deal only with the primitives needed to set the communication parameters, since most PC-based Forths make use of the ROM BIOS interupt calls to send and receive characters. Mr. Cooper's paper provides fine examples of how to create an `SKEY` and `SEMIT` word to send and receive characters in high-level Forth. Also, for my purposes, the reason for this lexicon was to achieve the higher baud rates needed to comply with the SONY control protocol.

The first few words in the file are simply to make the job a little easier. `WITHIN?` is my version of a word that goes by many aliases in other systems, so you may be able to use the equivalent from your system—taking care, of course, to see if the input requirements are the same. `BINARY` is self-explanatory; you may also find an equivalent in your system for this word. `SPLIT` is a kernel word in HS/FORTH, but I include it here for reference. `SAL` (shift arithmetic left) is also available as a code word in most systems, but not neccessarily by the same name. For those who don't have it, I have included a much slower, but useable, version.

The constants and variables defined are explained in Figure One.

With regard to the information needed to set up the 8250, the approach taken here is to calculate the required number from user-supplied inputs. The alternative approach is to use lookup tables for the bytes the chip needs. While reviewing information about the chip, however, it became clear that all the neccessary numbers could be easily calculated using real-world input that a user could intuitively understand.

The word `COM-PORT` does just this. It can be seen that the I/O address of com (i.e., communication) port 0 is at hex 3F8, or

COM_BASE as I have called it. Note, then, that all other com ports use an address exactly 100 hex bytes lower. Therefore, COM-PORT can calculate the address of a com port very easily by using the value of COM# times 100 (hex) and subtracting the product from the COM_BASE.

Next, by using the base address returned by COM-PORT plus an offset byte count, we can address any of the registers in the 8250 register complement. The word REGISTER does this. The usage then becomes simply 3 REGISTER to return the I/O address of the line-control register we are interested in. This language is clarified by using the constant LINE-CTRL instead of the literal number three.

A peculiarity of the 8250 is that two of the control registers, those used to store the baud rate divisor, are kind of hidden. They can only be changed if the seventh bit of the line-control register is set (i.e., high). The word BAUD! takes care of these details. A copy of the contents of the line-control register is put on the stack and also on the return stack.

The parameter stack's copy has its seventh bit set high by ORing hex 80 with it and storing the resulting number back into the line-control register. Now the first two registers will accept bytes as the baud rate divisor. This number is used by internal hardware to time how long an interval to use for a given number of bits per second; in other words, how much time to use for each bit.

BAUD! wants an integer on the stack for input, and this integer is split into two bytes by the word SPLIT. The upper byte is placed into register one and the lower byte into register zero. This completely abstracts the problem of setting the baud rate divisor. It's now all one word.

The primitives are now almost in place. In fact, the word LINE-CTRL! did not exist until I had completed the entire lexicon and reviewed my code. As is usually the case, I found that I had used a similar phrase in many word definitions. By extracting the common phrase from those words and giving it a name, I reduced the total amount of both source code and compiled memory while improving the readability of the final source code. LINE-CTRL! uses an input byte and a mask byte as input arguments, and simply changes the bits in the register by using the mask to AND all the bits you don't want changed and to OR the new bits into the old register contents.

This completes the complement of primitives needed to get on with making a user-interface lexicon, the actual words an end user will use.

High-Level Stuff

When I was thinking about how this lexicon should read to the end user, it occured to me that the English words we use for talking about serial communication parameters are perfect Forth syntax. This became my model. I wanted it to read like this, for example:

```
1200 BAUD
ODD PARITY
7 BITS
2 STOP-BITS
```

The first word needed is BAUD. It will use BAUD!, of course, but what numbers are stored in the baud rate divisor registers? A quick look at Figure Two will reveal a linear relationship between the numbers needed for a given baud rate and the baud rate itself. It is, therefore, a simple matter to find the constant required to derive the correct divisor: multiply the baud rate by the baud rate divisor, expressing both numbers as a 16-bit integer. For example, the table tells us that the value needed for a speed of 300 baud is 0180 hex or 384 decimal. If we multiply 300 * 384, we get a value of 115,200. This is our magic number, so in the code the constant is named MAGIC#. Now we can calculate any number in the table simply by dividing our MAGIC# by the baud rate.

The only problem with this is that 115,200 must be stored as a double-precision constant and, therefore, our dividing must be done with a mixed operator, one that divides a 32-bit integer by a 16-bit integer. In the Forth-83 Standard, the operator that does this is UM/MOD. I chose it for the sake of compatability with this paper. This operator is not quite what we need, since it returns a remainder with the quotient, but that is easily DROP-ed in the code. (Check your own system for an operator like M/ if you wish, and don't forget to remove the DROP in in the existing code.)

The word BAUD checks for invalid baud rates, i.e., those less than zero or higher than 57,600 bps. This is done effectively by

using a common Forth trick, that being the U< operator. We then calculate the divisor needed, SWAP DROP the remainder, and pass it on to BAUD!.

The rest is easy. All the other high-level words simply trap for bad inputs using ABORT" and then use a shift arithmetic left (SAL) operator to move the input numbers into the correct bit position. Then, with the correct mask byte, LINE-CTRL! puts the bits into the 8250's line-control register.

For parity, three constants are defined. NO is zero, ODD is one, and EVEN is two. These values control parity in the 8250.

Of interest is the fact that the 8250 uses the digits 0–3 to represent the number of bits that will be used. Since the chip designers graciously made the control-numbers sequence in the same order as the number of bits each number represents, we can calculate the proper parameter simply by subtracting five from the input. This allows us to write nice Forth statements like 8 BITS.

Conclusions

It can be seen from this exercise that the principles of Forth programming make a lot of sense when writing an interface for real-world devices. The concepts of small, reusable code blocks (words), coupled with the ease with which parameters can be passed between them, allows the programmer to create a syntax that is not only just right for the job but which is also completely integrated into the programming environment.

When interfacing to hardware, look for relationships between what the hardware needs and what the user wants to provide it. In this case, a lot of memory was saved by doing away with lookup tables and complex logic involving IF or CASE statements, simply by calculating the values needed.

I suggest that these words, or ones like them, become standard word names for setting up a UART chip. The behind-the-scenes details of how they work is thoroughly hidden to the end user, but the primitives could be easily modified to work for any UART. Frankly, given the fact that Forth is the langauge of choice for hardware interfacing, I am surprised there is not a standard wordset for dealing with I/O devices (except for disk-block words). Perhaps the ANSI group is dealing with that. Sounds like there is another article in there somewhere. Any takers?

Bibliography

Brodie Leo, *Thinking Forth*. Prentice Hall, Inc., 1981.

Jourdain Robert, *Programmers Problem Solver*. Brady Books (a division of Simon & Schuster.)

Brian Fox is an engineering technician with CFPL Television in Canada. He is the author of INTELECT, a text-driven data base written in HS/FORTH that allows users of a newsroom computer to talk to a character generator for high-speed election result displays. Brian has been using Forth for about five years.


```
( 8250 Communication parameter control     24NOV88  FOX )
( Written using HS/FORTH V3.8 with the Forth-83 overlay )

HEX
\ This version is true if n is within or equal to the limits
: WITHIN?  ( n lo-limit hi-limit -- ? )
  >R 1- OVER <  SWAP R> 1+ < AND ;

: SPLIT  ( n -- lo-byte hi-byte )
  DUP 00FF AND  SWAP 100 / ;

: BINARY 2 BASE ! ;
```


(Code continued on next page.)

```
(\ Check your system for it's own SHIFT ARITH-
METIC LEFT operator
: SAL     ( bit-cnt -- )
  0 DO 2 * LOOP ;      \ poor man's substitute

VARIABLE COM#    0 COM# !
3F8  CONSTANT COM_BASE
003  CONSTANT LINE-CTRL
1C200. 2CONSTANT MAGIC#  \ divide by BPS to
get divisor for 8250

BINARY
  11100111 CONSTANT PAR_MASK
  11111011 CONSTANT STOP_MASK
  11111100 CONSTANT BITS_MASK

HEX
: COM-PORT   ( -- com#-address )
  COM_BASE COM# @ 100 * -- ;

: REGISTER   ( n -- addr)
  COM-PORT + ;                          \ select
an 8250 register

: BAUD!      ( n -- )                   \ store
baud rate divisor
  LINE-CTRL REGISTER P@ DUP >R       \ stack
line-ctrl reg. value
  80 OR LINE-CTRL REGISTER P!          \ set
7th bit in the register
  SPLIT  1 REGISTER P!  0 REGISTER P! \ store
the bytes of divisor
  R> LINE-CTRL REGISTER P! ;         \ restore
line-ctrl reg. value

: LINE-CTRL! ( c -- )
  LINE-CTRL REGISTER DUP >R  P@ AND OR   R>
P! ;

DECIMAL
: BAUD    ( n -- )
 DUP  57600 U< NOT ABORT"  Invalid baud rate"
 MAGIC# ROT UM/MOD SWAP DROP BAUD! ;

: STOP-BITS ( n -- )
 1- DUP 0 1 WITHIN? NOT ABORT" 1 or 2 allowed"
  2 SAL  STOP_MASK  LINE-CTRL! ;

0 CONSTANT NO
1 CONSTANT ODD
3 CONSTANT EVEN

: PARITY
 DUP 0 3 WITHIN? NOT ABORT" Bad parity type"
  3 SAL  PAR_MASK  LINE-CTRL! ;

: BITS ( c -- )
  5 -  DUP 0 3 WITHIN? NOT ABORT" Bad bit
count!"
  BITS_MASK  LINE-CTRL! ;

EXIT

\ Examples of use of 8250 control words
  0 COM# !    300 BAUD   ODD PARITY  7 BITS
1 STOP-BITS
  1 COM# !   1200 BAUD  EVEN PARITY  7 BITS
1 STOP-BITS
  2 COM# !  19200 BAUD    NO PARITY  8 BITS
2 STOP-BITS
  3 COM# !  38400 BAUD    NO PARITY  8 BITS
1 STOP-BITS
```

Advertisers Index

```
HEX
3F8 COM_BASE ..... The base I/O address of the first serial
                   port in a PC-type computer.

3 LINE-CTRL ..... The offset in bytes from the COM-
                  PORT address to the 8250's line-con-
                  trol register.
1C200. MAGIC# ..... This number, when divided by the
                    number of bits per second, gives the
                    baud rate divisor.

Binary constants
PAR_MASK ..... Mask byte to select "parity" control bits.
STOP_MASK ..... Mask byte to select "stop-bits" control bits.
BITS_MASK ..... Mask byte to select "bits" control bits.

Variable
COM# ..... A user variable in HS/FORTH. It stores the
           com port number in use (zero is the first
           COM#).
```

Figure One. Constants and variables defined by the program.

Baud Rate	3F9H	3F8H
110	04H	17H
300	01H	80H
600	00H	C0H
1200	00H	60H
1800	00H	40H
2400	00H	30H
3600	00H	20H
4800	00H	18H
9600	00H	0CH
19200	00H	06H
38400	00H	03H
57600	00H	02H

Figure Two. Table of baud rate divisors for the 8250 UART.

May/June 1989

THE BEST OF GENIE

GARY SMITH - LITTLE ROCK, ARKANSAS

One of the 'Best of GEnie' columns to garner the most interest appeared in the September/October 1988 issue of *Forth Dimensions* (X/3). I think this is more than understandable, since that column featured the Real-Time Conferences on the GEnie Forth RoundTable. We *are* interested in the thoughts and philosophies of our fellow Forthers.

To recap that column briefly: I discussed Leonard Morgenstern's excellent Sunday night Figgy Bar, which addresses fundamental Forth coding issues. I must repeat, Leonard's knowledge and style make this a most pleasant opportunity to expand your ability to write Forth code. Thursday night is the regular Figgy Bar, where I attempt to maintain some sense of order amidst mayhem. No rule is the rule here, as long as the topics pertain to Forth. Some wonderful discussions result despite the din. One that comes to mind centered on VMS (Virtual Mass Storage) for FPC (Zimmer and associates' excellent F-83 for PCs). The exchanges were wonderful.

The stars of the Real-Time Conferences are our special guests. Guests since that issue of *Forth Dimensions* have included Mitch Bradley ("Forth in a Unix environment"), Larry Forsley ("Forth and the Future"), Dr. C.H. Ting ("Forth and Zen"), and Larry Forsley ("Forth in Publications"). I will conclude this conference series in the next issue with synopses of conferences that have featured George Shaw, Mike Perry, Randy Dumse, and Wil Baden.

It should be clear, if you are not participating in these conferences, that you are missing out on a lot of Forth knowledge and insight. As before, I will feature the guests' opening remarks for their respective conferences.

Mitch Bradley
August 1988
Staff engineer with Sun Microsystems and owner of Bradley Forthware.

<[Mitch]> Sun Microsystems: Leading supplier of technical workstations, has grown to $1 billion annual sales in six years. I've been there since 1982 (employee #50, now over 7500 employees).

Bradley Forthware: My home company. Supplies Forth for 680x0 machines (Atari ST, Macintosh, Sun, others) and SPARC machines (Sun, single board).

Past use of Forth at Sun:

- "Unofficial" bring up standalone diagnostics.
- A small number of Unix-based tools.

Current Forth work at Sun:

- Developing new Forth-based firmware for Suns; will probably become the standard firmware shipped with Suns sometime in the future.
- Forth is used as an interactive monitor and as a CPU-independent language for "plug-in" boot drivers (not Unix drivers; they're still written in C).

My (perhaps) controversial beliefs:

- Forth needs to "grow up."
- The "minimalist" Forth philosophy is responsible for Forth's relative lack of success.
- Screens suck.
- Forth's portability problems are due to disregard of reality.
- C has its advantages.
- Forth chips are no big deal.

Larry Forsley
October 1988
Director of the Institute for Applied Forth Research.

<[larry]> Forth and the Future. First, what is the future of computing? Procedural languages, non-procedural languages, expert systems, neural networks, artificial intelligence?

Perhaps we're heading for augmented intelligence, where the computer is an amplifier. Ah, convergence between Forth and the Future. Chuck Moore speaks of Forth as an amplifier for good and bad programmers and applications alike.

What is Forth? Who is Forth for? What does Forth do best? What do we want to do with it? And a paradox... "Forth is like the Tao. It is a way which is realized when followed. Its fragility is its strength. Its simplicity is its direction."

Given all this, I propose some trends I've observed and wish discussion upon. Unlike UNIX/C, Forth hasn't hit well with academia without students growing up to be employed like their parents. We lack Reaganomics as an economic engine. Without that engine, the military doesn't want us, either. If the military doesn't want us, we're not real; unlike well-fed physicists, and more like hungry biologists. This community has kept its minds clean, but hasn't yet had opportunity.

Dr. C. H. Ting
October 1988
Software engineer with Maxtor.

<[ting]> Let me start by quoting Dr. Lin Yu-Tang, a famous Chinese modern writer, "A man's speech should be like girls' skirts. None preferred." If have to, the shorter the better.

<[Gary]> Read the file PREZEN2.TXT to receive full benefit of Dr. Ting's topic intent. The notes follow:

Welcome to this Zen and Forth Conference. What I like to focus on tonight is the religious aspect of Forth. Zen is the religious development in China which shares many characteristics with Forth. I would like to use it to open our discussion.

Zen stresses simplicity. Enlightenment is not as complicated as traditional Buddhism leads you to believe. It is not in the documentation. It is not in the established practices. It is in yourself. You have it already. But you have to discover it yourself. It is also an experience, which can only be passed from mouth to mouth and from heart to heart, not by books or written words. It became an oral tradition.

Forth is the Zen of computing.

Background of Zen in China:

Over hundreds of years, Buddhism spread into China from India. Tons of literature was translated, most so badly that only the priesthood could make sense of it. Lots of temples and monasteries were established. Millions were led to believe they could attain enlightenment and a better second life by certain practices and rituals.

The Zen masters found that the best way to attain enlightenment was not through study of literature, not through established rituals, but by ridding oneself of foreign things. Meditation, hard physical labor, occasional loud noises, and sometimes a sharp blow on the head would precipitate the enlightenment.

Zen became very successful in China and Japan because it integrated the essences of Buddhism and Taoism. The more foreign Hindu influences were diluted, so Zen grew more naturally in the Chinese environment. It became the principal Far Eastern philosophy dealing with the meaning of life and provided a framework that individuals could find happiness and satisfaction in life.

Similarities between Zen and Forth:

- They present their subjects in the simplest ways, stripping away any irrelevant elements that hinder understanding the essence of the subjects. Simplicity is the most common trait.
- They broke away from massive establishments overloaded with church, documentation, practices, and, unfortunately, money.
- Enlightenment is not derived from established church, documentation, practices, or money.
- Enlightenment does require personal effort. When personal effort reaches a critical mass, enlightenment usually comes in a flash.
- They both started as oral traditions. Documentation was considered untrustworthy, dangerous, and superfluous.
- They attracted feverish and loyal followers, as well as opposition of similar intensity.
- They became fragmented as individuals perceived their own brands of truth. Standards committees were organized to iron out differences, but never quite succeeded.

Religious Experience in Forth Programmers:

The most intense form of religious experience is enlightenment in Zen and Buddhism, rebirth in Christianity, commitment in marriage, etc.

Most Forth programmers had very similar experiences that convinced them of the righteousness of Forth. It generally happens after the completion of a substantial project, with the full understanding of INTERPRET, figuring out what DOES> does, or when seeing 'ok' from a system one is building. Prior to that experience, he generally has spent a couple of months trying figure out why Forth works or why his Forth does not work.

Casual Forth users, like taking a course in Forth, generally do not have the intense period of study and exploration to warrant the enlightenment. Most of them drift away at the next wind.

Let's open up the floor and see how many people share a similar experience. I am particularly interested in at which stage you attained enlightenment, if you did. The next question is: Is this experience transferable to other people? Is this Forth disease contagious?

It seemed that Forth was contagious in '77–'80 period due to the explosive growth of FIG. It also seems that it is not contagious now. Anybody care to comment?

Why is Forth Right?

Why is Forth right, and why are all other computer languages wrong? Because Forth is the essence of computing. Then, what is computing?

Man created this machine, somewhat in his own image, logically. He tried, but not quite succeeded. Because he is too complicated to be cloned by mechanical or electronic media. He wanted this machine, he called it a computer, to do something useful; that is, things he would otherwise have to do himself. They include doing lots of addition, multiplication, logic operations, piloting an airplane, and guiding a missile. Because the machine is of lesser 'intelligence,' it must be given precise, unambiguous instructions.

The essence of computing is to give this machine precise, unambiguous instructions. Why is Forth right? Because it is the best vehicle to construct and to deliver precise, unambiguous instructions to this machine.

Forth consists of a set of instructions that we call words, which Man can use to make the Machine do what he desires, within the capability of the Machine. This is true for all other languages. But Forth delivers the instructions directly to the machine (it interprets), and new instructions can be constructed freely from existing instructions (it compiles).

In Forth, Man is not programming the Machine. He is designing a new Machine by adding new instructions to it, so that the Machine becomes a closer clone to his image. Forth is right because it allows the Machine to grow to be more like its master.

The Forth syntax is simple because the Machine understands it best, and it is not difficult for the Man to learn and to use. The Man gets the best satisfaction if he means what he says, because the Machine does exactly what he says. Simple syntax does not means weak syntax, as Forth syntax fully supports the classical control structures and modularity touted by the proponents of modern structured programming.

I like Bach's music, but I cannot play. So I programmed my computer to play his music for me. To program it became very tedious, because I had to enter every note to build a whole piece. I gave my computer a scanner so it scans the music and converts the score to notes it can then play for me. The computer reads the music, very similar to the way I read the music. It writes the notes to a file, just as I would have written them. I cloned a part of myself in my computer. You can do it in any language, but Forth let me do it in the shortest time.

(Continued on page 38.)

REFERENCE SECTION

Forth Interest Group

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

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

Board of Directors

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

Founding Directors

William Ragsdale
Kim Harris
Dave Boulton
Dave Kilbridge
John James

In Recognition

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

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

On-Line Resources

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

GEnie

For information, call 800-638-9636

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

BIX (ByteNet)

For information, call 800-227-2983

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

CompuServe

For information, call 800-848-8990

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

Unix BBS's with Forth conferences (ForthNet links)*

- WELL Forth conference
  Access WELL via CompuserveNet or 415-332-6106
  Fairwitness: Jack Woehr (jax)
- Wetware Forth conference
  415-753-5265
  Fairwitness: Gary Smith (gars)

PC Board BBS's devoted to Forth (ForthNet links)*

- East Coast Forth Board
  703-442-8695
  SysOp: Jerry Schifrin
- British Columbia Forth Board
  604-434-5886
  SysOp: Jack Brown
- Real-Time Control Forth Board
  303-278-0364
  SysOp: Jack Woehr

Other Forth-specific BBS's

- Laboratory Microsystems, Inc.
  213-306-3530
  SysOp: Ron Braithwaite

This list was accurate as of March 1989. If you know another on-line Forth resource, please let me know so it can be included in this list. I can be reached in the following ways:

Gary Smith
P. O. Drawer 7680
Little Rock, Arkansas 72217
Telephone: 501-227-7817

Fax: 501-228-0271
Telex: 6501165247 (store and forward)
GEnie (co-SysOp, Forth RoundTable): GARY-S
BIX (Bytenet): GARYS
Delphi: GARY_S
MCIMAIL: 116-5247
CompuServe: 71066,707
Wetware Diver. (Fairwitness, Forth Conference): gars
Usenet domain.: gars@well.UUCP *or* gars@wet.UUCP
Internet: well!gars@lll-winken.arpa
WELL: gars

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

(Continued from page 36.)

The computer grows steadily, like a child, but only one word at a time.

Larry Forsley
December 1988
Owner of DashFind Associates and publisher of the Journal of Forth Application and Research.

<[larry]> Forth is going on towards 20 years. For the first several, it was totally an oral tradition. FORTH, Inc. published some early manuals. Two papers appeared in the IEEE and astronomy literature, and all was silence until *Forth Dimensions*, FORML Conferences, the *Journal of Forth Application and Research*, and *Rochester Conference Proceedings*. Then *Dr. Dobb's Journal* and the big one... *BYTE* magazine 1980 and their first language issue, which happened to be on Forth.

As of two years ago, when last I counted, there were about 2,000 references to Forth in the literature. But are we writing what we're doing more than before? Are we writing to ourselves or to a larger group?

Early on, Elizabeth Rather worried that the Journal would take good authors away from the 'open' literature. Did that happen? Who reads about Forth in Dobb's? And, what about textbooks. Even with Brodie, Winfield, Haydon, Kelly and Spies, and Pountain, why isn't there a suitable college-level text that ties Forth into the rest of computing?

<[Gary]> Larry: I know you were approached on this. Will you please, for once and for all, respond to the non-performance of [the *Journal of Forth Application and Research*] this year. Where is it? What are your plans to make good on purchased volumes, and when?

<[larry]> Thanks for the chance to respond. *JFAR* V,2 will be going to the printer ... just before Christmas. It has papers by Dress (neural nets), Grossman (a solver for f(x)=0), Noble (on the death of Fortran), Roye (exception handling), Feucht (LISP and a new number system), and a bit more. We have been held up converting to an electronic system. We now use Ventura Publisher and have found the transition very painful. *JFAR* V,3 and V,4 papers are now being processed. I expect volume V to be finished by June '89.

CHAPTERS DOWN UNDER

JACK WOEHR - 'JAX' ON GEnie

We recently received an interesting letter from Lance Collins, the principal of Fifth Generation Systems, Ltd. in Australia, which we reproduce here in part:

"Dear Jack,
"Re: *Forth Dimensions* X/5, pg. 36 on email to chapter coordinators.

"I am the Chapter Coordinator for the Melbourne FIG Chapter and also the SYSOP of our OPUS BBS. Our problem in maintaining email contact with you is very simple. It's lack of money. My problem is that I am self-employed and don't have a kind employer to pay my telephone bills. The chapter members who can be relied upon are in a similar situation. (The academics in our chapters who do have access to things like USENET here are not reliable!)

"To put you in the picture about our chapter, I should start from how things were about a year ago. Meetings were held monthly in a small hall (they still are) and usually about ten people attended out of about 40 on our mailing list. Sometimes we talked about Forth, but mostly it was a general chat about micros. Nobody was actually doing anything in Forth. If my wife had not kept booking the hall and providing supper, the chapter would probably have folded. I was not doing anything in Forth because of a large software package I am developing in Dataflex. In May, I attended the first Australian Forth Symposium and realised there were a lot of people out there doing things in Forth who never have anything to do with FIG Chapters.

"With the promise of lots of support, I agreed to organize a second symposium in Melbourne this year. The first step was to legally incorporate our chapter and to set up a bulletin board to facilitate communication between the symposium organizers. (Incorporation because I found I was personally liable if someone had an accident at one of our meetings. The situations possible from a two- to three-day symposium for 200 people did not bear thinking about.) We have the board set up but the promised support for organizing a symposium did not appear (see note on academics, above). So there will not be an Australian Forth symposium in Melbourne this year.

"About last November, I decided to devote my efforts to getting the bulletin board going and to let monthly meetings die out if the members did not talk more about Forth. The results so far have been encouraging in two ways. First, thanks to a large pile of disks from Jerry Shifrin, there are many interesting files on the board and about 15 people have paid up for BBS membership; pretty good, as many have had to buy a modem as well. (Chapter fees are $20 for ordinary membership and $40 for full access to the bulletin board.) Significantly, five of these are from out of town or interstate. (We have never before had members outside of Melbourne, as we had nothing to offer.)

"Secondly, by laying down the law at chapter meetings, we are talking about Forth again, people are showing their Forth machines at meetings, and attendance is up to 15 or more. The numbers may seem small to you but we were starting behind scratch, as we had probably driven a number of people away in previous years. *[Frankly, Lance, those numbers sound great to us!]*

"Now we need some help. I am unhappy about the small amount of chat on [our FIG Chapter computer bulletin] board. We need someone to talk to. Seems to me, it is critical that we become part of a wider Forth conference so that our members can see what is going on over there and that they also can have an input..."

Lance Collins
65 Martin Road
Glen Iris, Victoria 3146
Australia
BBS (03) 299-1787
Voice (03) 299-2009
When dialing from U.S.A., replace the (03) with (61 3).

Lance's letter in interesting in several aspects. First of all, Lance's experience indicates that the general decline of what we might call "clubbiness" in the Forth community is not a phenomenon limited to North America. Observation indicates that it is difficult to draw people to computer meetings in an era of our societies when computers are extremely easy to come by. Possibly the next generation of Forth chips (SC32, 32-bit RTX) will bring 'em out again in droves to see and touch the new hardware.

Secondly, our personal experience at the Denver FIG Chapter coincides with that of Lance, in that a computer BBS is a powerful communication tool helping to keep the local Forth community from dispersing to the four winds. In Denver, our Forth-83 class meets weekly, but the chapter meetings are irregular. We all keep in touch meantime via the RCFB.

Last but not least, it is disappointing to hear that there will be no Australian Forth

Symposium this year. It seems that Forth and the careers of professional Forth programmers are booming while organized Forth social activity wilts on the vine. Do we no longer need each other?

The Talmud, the 2200-year-old collection of Jewish aphorisms, says, "Student, get thyself a companion in study!" Perhaps our generation of Forth programmers has graduated to success and we no longer feel the longing for fellowship, exchange, and mutual edification that we once felt, and have grown comfortable limiting that exchange to the BBS medium.

If that is the case, our duty is clear: it is time to induct the next generation of Forth programmers into the fold. Where, please tell me, where are we reaching out to the millions of computer-aware youth of today?

Jack Woehr
FIG Chapter Coordinator
well!jax@lll-winken.arpa
JAX on GEnie
SYSOP, RCFB: (303) 278-0364

(Continued from page 12.)

- If the mode is false (0), any key except the space bar, return, escape, and up-arrow will exit the program.

If you love complexity and high performance, you could add the property of typing over the display character and correspondingly updating the RAM and byte display.

Happy programming!

Allen Anway is a computer coordinator at the University of Wisconsin at Superior. His eldest daughter (of three) is a third-generation physicist.

(Screens continued from page 11.)

```
    SCREEN # 041
( # 041    ( Don't use this program! )
( ram-address ---- )  HEX -1 MODE !
: VVDUMP BASE @ HEX SWAP C SWAP BEGIN
    BEGIN BEGIN [ >R >R >R >R >R >R ]
    CLEAR B LAYOUT
    XKEY DUP DUP 1B = SWAP B = OR
IF ." ENTER" -LF -LF BS DROP
            4 HPOS ! 14 VPOS !
    BEGIN XKEY CASE
 8 OF <<                              ENDOF
1B OF >>                              ENDOF
 A OF vv                              ENDOF
 B OF ^^                              ENDOF
 D OF 108 - [ R> R> R> ] REPEAT       ENDOF
BL OF       [ R> R> R> ] REPEAT       ENDOF
1B OF 0 16 HVTAB CEOL
        2DROP BASE ! EXIT             ENDOF
      DIGIT'   0 ENDCASE REPEAT
ELSE DUP D =
  IF DROP 108 - ELSE BL -
    IF 2DROP BASE ! EXIT THEN THEN
THEN REPEAT ;

DECIMAL
```

(Continued from page 4.)

other of them will prove useful. Betancourt's approach, especially his lazy variables, will appeal to some; Yli-Nokari's method will be easy for F83 users to test; and Hayes' scope-based technique may be the most clean and versatile.

If you already have local variables and flatly don't want to know any more about them, turn to "Forth Needs Three More Stacks." I don't know *anyone* who has five stacks yet—who knows what it could lead to?

* * *

The flyer announcing this year's Rochester Forth Conference on industrial automation (June 20–25 at the University of Rochester) contained an interesting note from conference chairman Larry Forsley. The conference schedule includes Dr. Sergei Baranoff from the Leningrad Institute for Informatika, with a speech titled "From Russian, With Forth." If you haven't heard, Forth is rumoured to have made significant inroads behind the Iron Curtain—possibly, in part, because of its capital performance in limited address spaces. Dr. Baranoff is the author of a 1988 Forth textbook in Russian, about which Larry's flyer announces, "The print run of 100,000 copies sold out in two weeks, making this the most popular book on Forth written." Move over, *Starting Forth*, the Russians are coming.

—Marlin Ouverson
Editor

(Continued from page 15.)

```
: ARGUMENT
  CREATE 2* C, ;CODE ... ;
```

An interesting note is that the `;CODE` part of `LVARIABLE` should be nearly identical to the user variable code, except `UP` is changed to `'LFRAME`.

When comparing the execution speeds of the second and third versions of `HANOI`, the latter was about 20 percent faster. If `ARGS` were written as above, the second should be as fast or faster than the third. Unfortunately, I was unable to test it; anyone care to comment?

The numbering scheme starts from one instead of zero. This convention was picked (controversial pun intended) because the numbering in stack diagrams—and in human minds—also starts from one.

Since one of the design goals was not to have complicated state-smart words, compiler security was not implemented. This means that `LOCALS` and `ARGS` can be used without really declaring them. In this case, the closest upper-level frame is accessed, i.e., the `ARGS` or `LOCALS` of the calling word. This is considered a feature, not a bug.

Finally, it seems that local variables are not very useful in everyday work, since we already have the stack for temporary values. Especially the words `L1 ... L8` and `@1 ... @8` seem not to be very useful. However, when there is need for large amounts of temporary data, the local variables come in handy.

References

[Bow82] S.A. Bowhill: "Fast Local Variables for Forth," *FORML Proceedings 1982.*

[Bar82] Paul Bartholdi: "Another Aid for Stack Manipulation and Parameter Passing in Forth," *1982 Rochester Forth Conference on Data Bases and Process Control Proceedings.*

[Mor84] Leonard Morgenstern: "Anonymous Variables," *Forth Dimensions* (VI/1).

[Ros87] Peter Ross: "Local Variables," *Forth Dimensions* (IX/4).

Jyrki Yli-Nokari wrote a Unix guide in 1985 and is a consultant in a major Finnish software house, but he claims Forth as his first love. As a 1981 school project, he wrote a Forth-79 system and utilities for the PDP-11; his master's thesis was a multi-user, multi-tasking Forth-83 for the 6809.

(Continued from page 28.)

to list his cases without worrying about how many there are. Then, when he is done, he can count them and pass the count to `CASES`. It also allows nesting of the case statement inside another case statement or conditional or loop, without restriction. Also, by virtue of the condition stack, the case statement can be used in interpretation mode.

Summary

By adding three stacks to a Forth system—an auxiliary stack, a condition stack, and a case stack—we can eliminate nine primitives; eliminate the need for branching; and eliminate the restriction on the use of conditionals, which then become regular Forth words that can be used individually to define other words and can be used in interpretive mode. We can then have a simple implementation of conditionals and case statements, and can reduce the potential for errors.

Dr. Ayman Abu-Mostafa has implemented these ideas in his obect-oriented Forth (see "Letters," this issue) on a Prime minicomputer at California State University.

FIG CHAPTERS

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

U.S.A.

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

- **ALASKA**
  Kodiak Area Chapter
  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. Forth, AI
  12 Noon Tutorial, 1 p.m. Forth
  South Berkeley Public Library
  George Shaw (415) 276-5953

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

  Sacramento Chapter
  4th Wed., 7 p.m.
  1708-59th St., Room A
  Tom Ghormley
  (916) 444-7775

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

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

  Stockton Chapter
  Doug Dillon (209) 931-2448

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

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

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

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

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

- **GEORGIA**
  Atlanta Chapter
  3rd Tues., 6:30 p.m.
  Western Sizzlen, Doraville
  Nick Hennenfent
  (404) 393-3010

- **ILLINOIS**
  Cache Forth Chapter
  Oak Park
  Clyde W. Phillips, Jr.
  (312) 386-3147

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

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

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

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

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

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

- **MICHIGAN**
  Detroit/Ann Arbor Area
  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

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

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

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

- **NEW YORK**
  FIG, New York
  2nd Wed., 7:45 p.m.
  Manhattan
  Ron Martinez (212) 866-1157

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

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

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

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

- **PENNSYLVANIA**
  Villanova Univ. FIG Chapter
  Bryan Stueben
  321-C Willowbrook Drive
  Jeffersonville, PA 19403
  (215) 265-3832

- **TENNESSEE**
  East Tennessee Chapter
  Oak Ridge
  2nd Tues., 7:30 p.m.
  Sci. Appl. Int'l. Corp., 8th Fl
  800 Oak Ridge Turnpike
  Richard Secrist
  (615) 483-7242

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

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

  Houston Chapter
  3rd Mon., 7:45 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
  D.C. & Northern Virginia
  1st Tues.
  Lee Recreation Center
  5722 Lee Hwy., Arlington
  Joseph Brown
  (703) 471-4409
  E. Coast Forth Board
  (703) 442-8695

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

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

INTERNATIONAL

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

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

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

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

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

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

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

  Toronto Chapter
  John Clark Smith
  PO Box 230, Station H
  Toronto, ON M4C 5J2

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

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

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

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

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

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

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

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

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

SPECIAL GROUPS

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

1989 ROCHESTER FORTH CONFERENCE
ON
INDUSTRIAL AUTOMATION

The Conference is sponsored by the Institute for Applied Forth Research, Inc., and will be held June 20-25, 1989 at the University of Rochester in Rochester, New York in cooperation with the Department of Physics and Astronomy, of the University of Rochester and the IEEE Computer Society. This is the ninth Rochester Forth Conference and it is sponsored by Dash, Find Associates, Harris Semiconductor, Mikrap, Miller Microcomputer Services, and the NASA Goddard Spaceflight Center.

Special Guest Lecturer

Dr. Sergei Baranoff, Leningrad Institute for Informatika, Leningrad, USSR,
"From Russia with Forth"
Dr. Baranoff is the 1988 author of the first Forth textbook in Russian. The print run of 100,000 copies sold out in 2 weeks, making this the most popular book on Forth written.

Invited Speakers

Don Berrian, Chief Engineer,Varian/Extrion Beverly Operation, Beverly, MA,
"Forth-based Control of an Ion Implanter"

Klaus Flesch, VP Engineering, FORTH Systeme – Angelika Flesch, Breisach, FRG,
"SwissFORTH, A Development and Simulation Environment for Industrial and Embedded Controllers"

Bob McFarland, President, Digalog, Ventura, CA,
```
": Cellmate/Toolbox
    Hardware/Software
    Workstation/Language DOES>
    Automotive/Aerospace
    Powertrain/Vehicle Development/Testing ;"
```

Jim Reda, VP Engineering, VIDEK, Rochester, NY,
"An Application Specific Machine Vision System"

Dean Sanderson, Chief Programmer, Forth, Inc., Hermosa Beach, CA,
"Events and Objects: Industrial Control by Hierarchical Decomposition"

Additional Presentations

- Harris Semiconductor will present a seminar including papers by current users of the RTX 2000.
- There will be vendors exhibits, demonstrations and poster sessions.
- Vendors are welcome. Please contact us.

HOUSING and TRAVEL

Registration, hotel or dormitory housing call (716)-235-0168.

USAir is the Conference airline and Stewart and Benson are the Conference Travel Agents.
Unrestricted fares **40%** off coach
restricted fares **5%** off supersaver or best rate available.
Call Ms. Barbara George collect at **(716)-244-9300**.

MORE INFORMATION

Lawrence P. Forsley, Conference Chairman
Institute for Applied Forth Research, Inc.
Box 100 • 70 Elmwood Avenue
Rochester, NY 14611

Voice: (716)-235-0168
Fax: (716)-328-6426
Email: L.Forsley on GENie; LFORSLEY on BIX and DELPHI

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

Second Class
Postage Paid at
San Jose, CA