$10 Volume XVII, Number 3 September 1995 October

FORTH DIMENSIONS

Extending CASE

Sets, Stacks, and Queues

Bounds Checking for Stacks

Nanocomputer Optimizing Target Compiler: the processor-independent core

Contents

Features

Departments

Editorial

I'd like to thank the writer of the letter on the facing page (which we have titled "Forth vs. notForth," although its author might have preferred "FIG vs. Forth"). Forth Dimensions *welcomes critical input that might further our community's understanding of Forth, of itself, and of its relationship to the rest of the world. This letter, in particular, raises some specific points to which readers are invited to respond. My reply here aims at the more general issue...*

There has long been an interesting dichotomy in the responses of Forth users to those who ask about its lack of one feature or another. On the one hand, Forth minimalists reply with something like, "You don't need it" or "Forth already has that." The first retort tells the potential Forth user that his perceived need doesn't exist, that we understand his problem better than he does (which may sometimes be true, but it's tactless and blunt as a marketing approach). The second inflates some element of Forth beyond proportion or demonstrates limited understanding of the topic, as when telling someone that Forth "already is object oriented."

On the other hand are those who Mr. Kloman (and he certainly is not alone) seems eager to dismiss. They say, "Forth can do that!" and proceed to create systems that do so—whether it be bounds checking, heap managers, or genuine object orientation. Performance, size, the support of a reliable vendor, and the availability of professional programmers trained on such systems all are apparently irrelevant, as long as the point is proved. Some minimalists say those resulting systems aren't Forth at all, but examples of application-specific languages or mutations of Forth into something else.

Which approach exemplifies the true Forth?

There is a point in *Fiddler on the Roof* when two people are arguing and the protagonist agrees with both. Another person chimes in, "But Tevya, they can't both be right." To which he responds, "You, too, are correct!" Wisdom would suggest that the answer lies not in making this an either/or debate with one right and one wrong answer for every programmer and every situation. Nor is a properly general solution likely to be found in a dilute compromise.

For that reason, as well as for their inherent interest, we welcome to these pages debate, critical thinking, and alternative approaches. These can influence us to think about Forth in new ways, or can serve as valuable reminders of Forth's inherent strengths. Neither I nor this magazine, under my stewardship, endorse a minimalist or maximalist (or static versus evolutionary) view of Forth. We simply attempt to publish the best of the useful and interesting material submitted. So I encourage those who sympathize with Mr. Kloman not to drop an explanatory note on the heels of their departure, but instead to remain and contribute their opinions and experience, to engage with us in the enterprise of shepherding Forth into the future.

I do suspect, though, that the Forth community must adapt, if only because the rest of the programming world has changed, and continues to change. And if the Forth philosophy is to continue to have a relevant voice, we must thoroughly understand contemporary programming practices, and how they relate to Forth. If we are to adequately address the expectations of employers, Forth programmers, developers, educators, and computer scientists, we must understand their expectations and be able to address them expertly.

—Marlin Ouverson
FDeditor@aol.com

Forth Dimensions
Volume XVII, Number 3
September 1995 October

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

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

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

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

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $40/46/52 per year by the Forth Interest Group, 4800 Allendale Ave., Oakland, CA 94619. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621-0054."

Letters

Challenged by Macros

I found Wil Baden's article on "Macro Processing in Forth" (*FD* XVII/1) quite handy and useful. I've been a macro-processor fan longer than a Forth fan, and do quite a bit in M4 (the UNIX macro language). I was skeptical of macros in Forth, as I tend to feel they are a crutch, particularly in C. I worked for a brief while for Larry Rossler, who, along with Steve Johnson, contributed mightily to the growth of C in the 70's and 80's. Larry felt the C pre-processor was an absolute mistake. He asserted, and I later demonstrated to myself, that #include is totally unnecessary, and that #define should be limited to those occasions where a mnemonic constant is sufficient.

Therefore, I was skeptical of the need for macros in Forth, thinking them a crutch in any language. Reading Wil's article carefully, I found both support and challenge for my views. Challenge sufficient that I've come to accept the place of macros in Forth, and am working on their use in the more general-purpose text processing, data analysis work that I usually find myself. Challenged further, so that I'm working on an idea I call the *text multiplexor*, or "Tex Mux" for short. Using Wil's basic idea, the controlling string multiplexes the controlled input strings onto the output. I'll need more time to explain, so another article or two should be forthcoming. It's based on fusing three things:

- Wil's macros
- Mills and Linder's use of text queues
- synergy with C's standard I/O

Mitch Bradley's "Yet Another Interpreter Organization" in the same issue was quite good as well. I'd seen the code from connections in Rochester from the mid 80's and am moving to implement it in my ANS-Forth. Mitch's `TH` is a more elegant, if not robust—in my estimation—approach to the "hex problem" than Wil's `0x`. But that's the proof to me of the value of macros. They don't belong everywhere.

Keep the magazine coming, Marlin!

Thanks,

Marty McGowan
Whippany New Jersey

Marty McGowan's article "Sets, Stacks, and Queues" appears in this issue. —Ed.

Forth vs. not-Forth

You may well believe that Forth programmers have drifted away from the Forth Interest Group (FIG) because of the recent recession years. I ask you to consider that Forth programmers did not drift away from FIG, but that FIG drifted away from Forth programmers. I believe there are many, many programmers around the world who, like myself, program in the Forth language whenever it is the appropriate language to use (which is most times for skilled Forth programmers).

I have been a Forth programmer since the original article in *Scientific American*. Forth is the main programming language I have used for many years. Assembly language is the second. I write Forth cores in assembly language. I have written cores for many processors and computers. But I have little interest in FIG and do not read FIG's publications. Here is the reason why.

Forth originated from the need to have an unlimited programming medium (it was originally written in a high-level language that was itself far too confining). It was designed to inherently encourage programmers and operators to be intimately close to the programming language, the hardware, and the data. Of course, the use of such an unlimited medium requires the full understanding of all three.

And this is where FIG parted from Forth. FIG took up the challenge of such things as object-oriented programming, type checking, etc. But the purpose of the use of these things is to separate the programmer and the operator from the programming language, the hardware, and the data of the system. Because these things are the counterpart of Forth, they should never be an extension of Forth. Other programming languages are available for those who need to be separated from the system. The C collection is an example of the present fad. Unfortunately, the main topics in *Forth Dimensions* became how to make Forth into these other programming languages; how to make the programmer less intimate with Forth, hardware, and data.

To further illuminate the philosophical difference between Forth and what is not Forth, I offer a few ideas:

- Forth programmers limit and manage the source and path of data so that there is no need for type checking, etc. Each type can never get into the wrong path. Objects are handled by their own code and do not need to be identified. Each path is inherently able to handle any data that can get into it.
- And Forth programmers use the inherently easy debugging checks of Forth so that runaway programs don't happen. There is no need for "bounds" checking. Forth programs don't run away because Forth programmers write closed paths.
- Forth programs run fast because there is no need for type checking, definition checking, etc. The programmer has written and debugged the paths so that run-time checking is not necessary.

(Continues on page 37.)

Sets, Stacks, and Queues

Marty McGowan
Whippany, New Jersey

What More Can Be Said?

We all know about stacks. What more can you say about stacks that hasn't already been said? The adoption of ANS Forth has spawned discussions about stacks other than the fundamental data stack and return stack. Rather than floating-point or compiler or exception stacks, let's discuss using stacks in software applications. And while we are at it, we'll include stacks in the more general realm of *sets* and *queues*. These data types—sets, stacks, and queues—are all collections differing only in their access method. Stacks have the LIFO property where items are stored last-in, fetched first-out. Queues have the FIFO property: first-in, first-out. Let's say that sets have the AIRO property: any-in, random-out. The need to use one of these types is based on the application.

It is worth reviewing for just a moment. Stacks are used in Forth and other programming languages to isolate functions and communicate data between them; queues are used in process control applications, particularly to manage tasks in operating systems; sets are used in relational data tables, where order isn't explicit. Some people have criticized Forth because of the many stack operations, as stack operations (in a pure stack) may only take place on top. Forth allows direct manipulation of many other stack items than the top. More words have been said on this subject than is necessary. Similarly, in operating systems, queues are examined and manipulated at places other than the ends. Rather than be too rigorous, let's take a practical approach. We will implement a pure, or simple, set of operations, but with a few hooks so we can traverse all the elements of each type.

My motivation for this article comes most recently from the "two stacks" discussion in comp.lang.forth and, more deeply, from an article, "Data Structured Programming: Program Design without Arrays and Pointers" by Harlan Mills and Richard Linger.[1] Implied by the title, Mills and Linger believe and discuss how many programming errors are introduced by misuse and overuse of arrays and pointers. Their suggestion is to use a more appropriate data type: a set, stack, or queue. At this point, you might be skeptical about replacing arrays and pointers with sets, stacks, and queues. Mills' and Linger's case is more clearly directed at the procedural languages. As an example, they show a Pascal statement which contains much potential for error:

```
a[i] :=  b[j+k]
```

Two arrays with three separate indices are being managed, each having their potential for error. As a Forth programmer, you are less likely to attempt this than your C or Pascal counterpart. But we're always in a position to learn from others. So, what do these types of sets, stacks, and queues have to offer the Forth programmer? First, they substantiate Forth's claim of simpler implementations. Next, like Forth, these types enforce the idea that simple tools can change the way we look at problems. I've a feeling, which I'll pursue in another article, that properly used, these types relieve some of the pressure on the return and data stacks. In the implementation here, the words are designed to allow arbitrary growth for members of the type. For example, stacks may be arbitrarily deep, queues arbitrarily long, and sets arbitrarily large. This comes at a performance penalty; the idea is that, during program design, a new type needn't be sized until sufficient use tells us what to expect; then it may be coded with a fixed-size type instance, which may be more efficient. Practicality isn't always machine efficiency.

Mills and Linger show how to declare and use the three new types in a Pascal-like syntax that should suggest where we're going:

```
set r of T;
...
member(r) :=  x;
y := member(r);

stack s of T;
...
top(s) := x;
y := top(s);
```

[1] "Data Structured Programming: Program Design without Arrays and Pointers," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, p. 192–197.

```
queue q of T;
...
back(q) := x;
y := front(q);
```

In a Forth implementation, we expect to see similarities and differences with a Pascal or C version. First, the major difference is that the Forth sets, stacks, and queues will be typeless. When we create a set, for example, the only thing the set will contain are cells of an unknown type. As with other types in Forth, the type of the set is up to the user. For example, we might have a set of queues. The set stores and retrieves arbitrary members, so we will need Forth words to accept and retrieve set members. We will also need a word to declare, or create, sets. Similarly for the stack, where the "top" is the only accessible member. And in the case of the queue, items are stored at the back and fetched from the front. We draw on Forth words fetch and store (`@` and `!`) to suggest the new names:

```
set:   (   compile: {parse} --    )
       ( execution:        --  set )
set!   ( n  set  -- )
set@   ( set -- n )

stack: (   compile: {parse} -- )
       ( execution:   -- stack )
stack! ( n stack -- )
stack@ ( stack -- n )

queue: (   compile: {parse} -- )
       ( execution:  --   queue )
queue! ( n queue -- )
queue@ ( queue -- n )
```

The operations are entirely regular, consistent with the core Forth words (`:`, `!`, and `@`). The type names with a trailing colon (`:`) parse a word at compile time, which at execution time leaves its address on the stack. Type names with a trailing exclamation (`!`) expect a value and the address of an instance, then store the value in the instance (not the address). Type names with a trailing at-sign (`@`) fetch a member of the type according to the rules of the type: FIFO, LIFO, or AIRO. Similar to the Forth data stack, but different from the memory operation, the side effect of the ...`@` operation is to remove the value from the instance. (E.g., `set@` removes the next item from the set, leaving it on the Forth data stack.)

At this point, the list of operators might be complete, but we're being practical, so two more operators are useful:

```
empty? (  set | stack | queue -- flag )

x-link (  t-a  t-b  --  )
\  exchanges identical types
```

`Empty?` returns *true* when the type has no members, *false* if occupied. For example:

```
set: test-set
test-set empty?  ( is TRUE )
```

Mills and Linger suggest defining the sparest list of operations, which seems a good rule. We'll see how to use these two utility words to traverse instances of sets, stacks, and queues. So applications like counting, summing, and printing which might be "built-in" are better left to the user. We'll take these up in a later section.

Design Goals and Objectives

Without getting carried away, the code should be as sparse as possible. One compromise I made was the use of the word `link`, which is used as a noun here. A node is replaced with its "link" on the stack, where a link is the forward pointer from one node to the next. Simply, it's:

```
: link   ( node -- node next ) dup @ ;
```

Nodes are two-cell pairs, where the first cell is the link and the second cell holds the value. I was carrying around `dup @` in the places where `link` was the idea. In debugging, I discovered I'd made a mistake in `queue@`. I'd originally coded `dup dup @`. The queue never emptied!

Sets and queues are similar in that they are maintained as *ring* types. A ring is a closed list, where the last node points to the first, which is the fetch point. Also, the ring pointer points at the last item, which is the insertion point for the queue. This is a well-known trick for ring types. Sets are different from queues because the order of fetching set elements can't be reliably predicted. This implementation simulates the random behavior of the set by moving the end pointer after fetching an element. The stack is implemented as a null-terminated linked list. When elements are fetched, they are removed from the type instance (successive fetches return different elements).

Another goal we have here allows any type instance to grow indefinitely. Sets, stacks, and queues will "never" overflow. This means we don't have to declare an initial size for each instance. How is this achieved? A single underlying freepool manages the cell-pairs of all types. A two-cell node is either taken from the freepool or allocated from the Forth dictionary. When an element, or cell-pair, is fetched and removed from the type instance, its two-cell node is returned to the freepool for later re-use.

Instances may be tested for being empty by the word `empty?`. The implementation uses a hidden value, rather than zero, to indicate an empty instance. You may want to have the value zero in sets, stacks, and queues. I could be persuaded that no useful item may be zero. For example, in a priority queue of tasks or processes, the interval to the next task may be zero, but that zero is probably better used as a value in another two-cell node, where one value is the time interval and the other is the task. I felt it better to use the hidden value as an empty sentinel rather than zero. Let's say it's open to discussion.

In order to non-destructively examine sets, stacks, and queues, the `x-link` ("cross-link") word allows swapping pointers to like type instances. The typical approach is to swap pointers between an empty type and the type of interest, which makes the empty pointer now point to the data and the pointer of interest an empty type. Then, successively fetching from the temporary instance and restoring in the type of interest allows inspection of the individual values.

Figure One shows an empty ring and an empty stack. Remember, sets and queues are implemented as rings. They have the property that when the last pointer points to itself, the ring is empty. The null value {0} may not be zero, but indicates the value is of no interest. Further attempts to fetch items from the empty ring return a value, after testing by `empty?`, of *true*. Figure Two shows an occupied ring. Following the insertion code shows the value is stored in place of the {0}, and a new node becomes the "last" after the current last. The first node is always the one after that. Figure Three shows the special freepool as a possibly non-empty, singly linked list of two-cell nodes. It is accessed as a stack.

Code Inspection

Sets, stacks, and queues are implemented in the code of Listing One. The words `INTERNAL`, `EXTERNAL`, and `MODULE` were invented (or discovered) by Dewey Val Shorre (*Forth Dimensions* II/5). They are something like:

```
: internal      latest  >link @ ;
: external      latest  >link ;
: module        ! ;
```

in a non-ANS Forth definition. Simply define them as no-ops in your system if you are willing to avoid using the words between `INTERNAL` and `EXTERNAL`. Word definitions (and variables, constants, etc.) between `INTERNAL` and `EXTERNAL` are available to the `MODULE`, but are otherwise invisible to later words in the dictionary. Words between `EXTERNAL` and `MODULE` are globally visible, unless some other wordlist restriction is in force. In Val Shorre's implementation, modules nest. I've seen suggestions how these words may be defined in ANS Forth, but I'd like to make sure they may indefinitely nest on one hand, and not be hemmed in by a wordlist limit. In the following discussion, the words `INTERNAL`, `EXTERNAL`,

Figure One. Empty ring with pre-fetched node (left); empty stack's null or zero pointer.

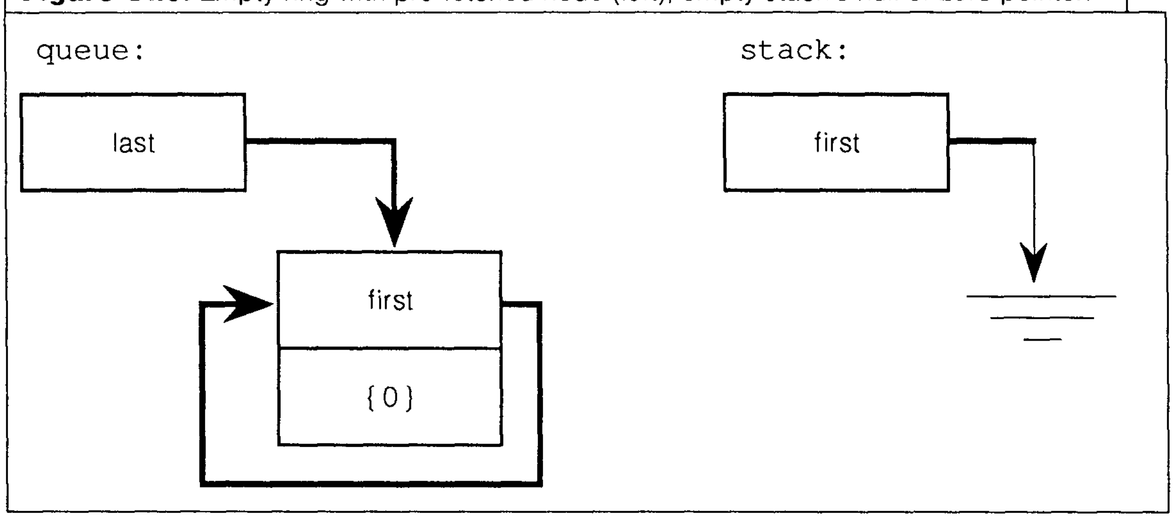


Figure Two. Occupied ring with insertion (queue or set).

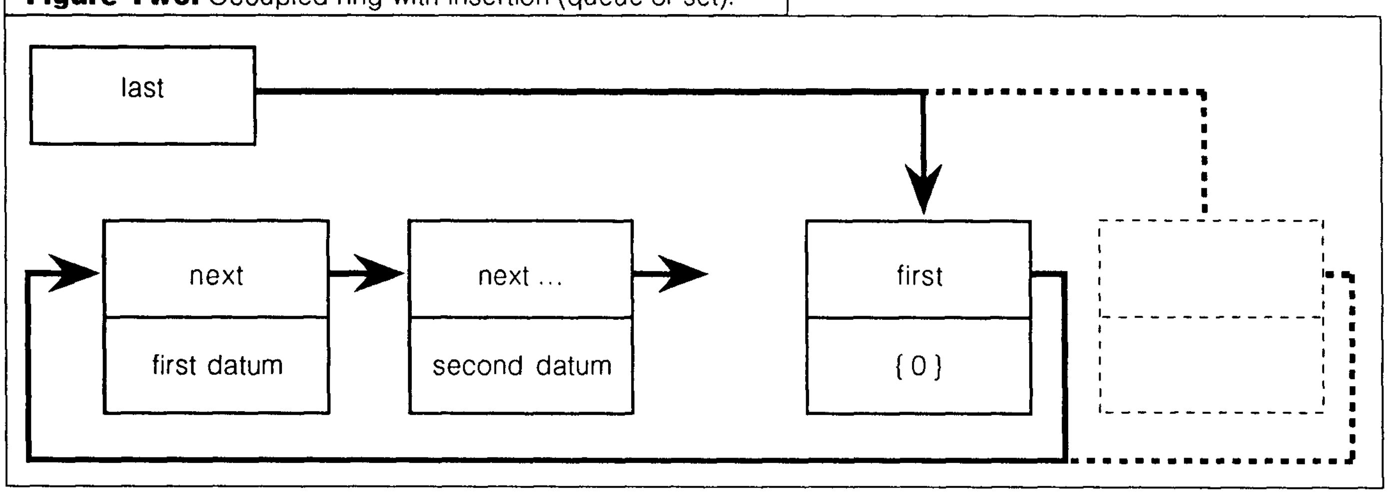
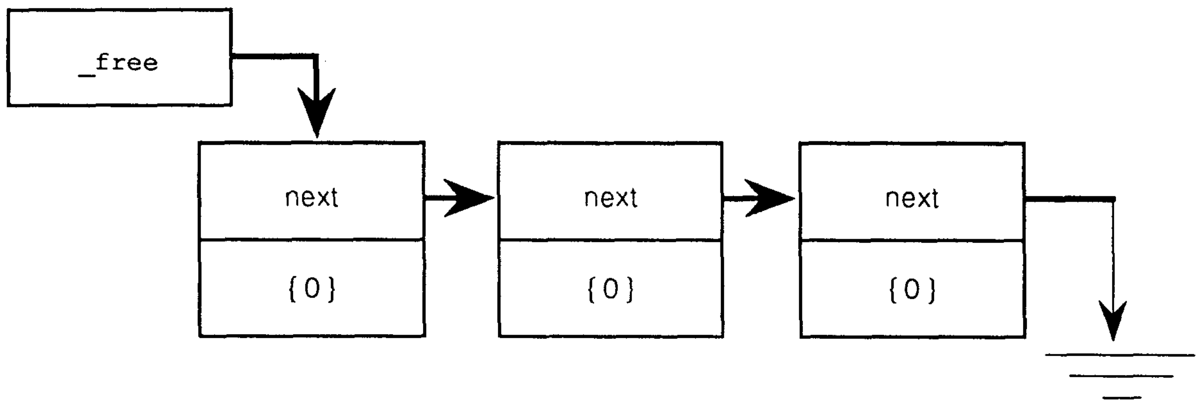


Figure Three. Freepool (stack access).



Listing One. Sets, stacks, and queues source.

```
( Sets, Stacks, and Queues -- Marty McGowan 950601)

INTERNAL

variable _free 0 _free !
: >free _free @ over ! _free ! ;
: free> _free @ dup if dup @ _free ! else drop here 2 cells allot then ;

: link      dup @ ;
: link@     link link rot ! ;
: link!     2dup @ swap ! ! ;
: _stack@   link cell+ @ swap link@ >free ;
: set++     @ 2dup = if @ then dup cell+ @ rot cell+ ! swap ! ;

EXTERNAL

: empty?   _free = ;     \ use hidden value, rather than 0
: x-link  link rot link rot rot ! swap ! ;

: stack:   create 0 , ;
: stack!   swap free> tuck cell+ ! swap link! ;
: stack@   link if _stack@ else drop _free then ;

: queue:   create here cell allot free> tuck dup ! ! ;
: queue!   tuck @ cell+ ! free> tuck over @ link! ! ;
: queue@   link link = if drop _free else @ _stack@ then ;

: set:   queue: ;
: set@   queue@ ;
: set!   tuck queue!  link link set++ ;

MODULE
```

and MODULE are used as reader's guides to the code.

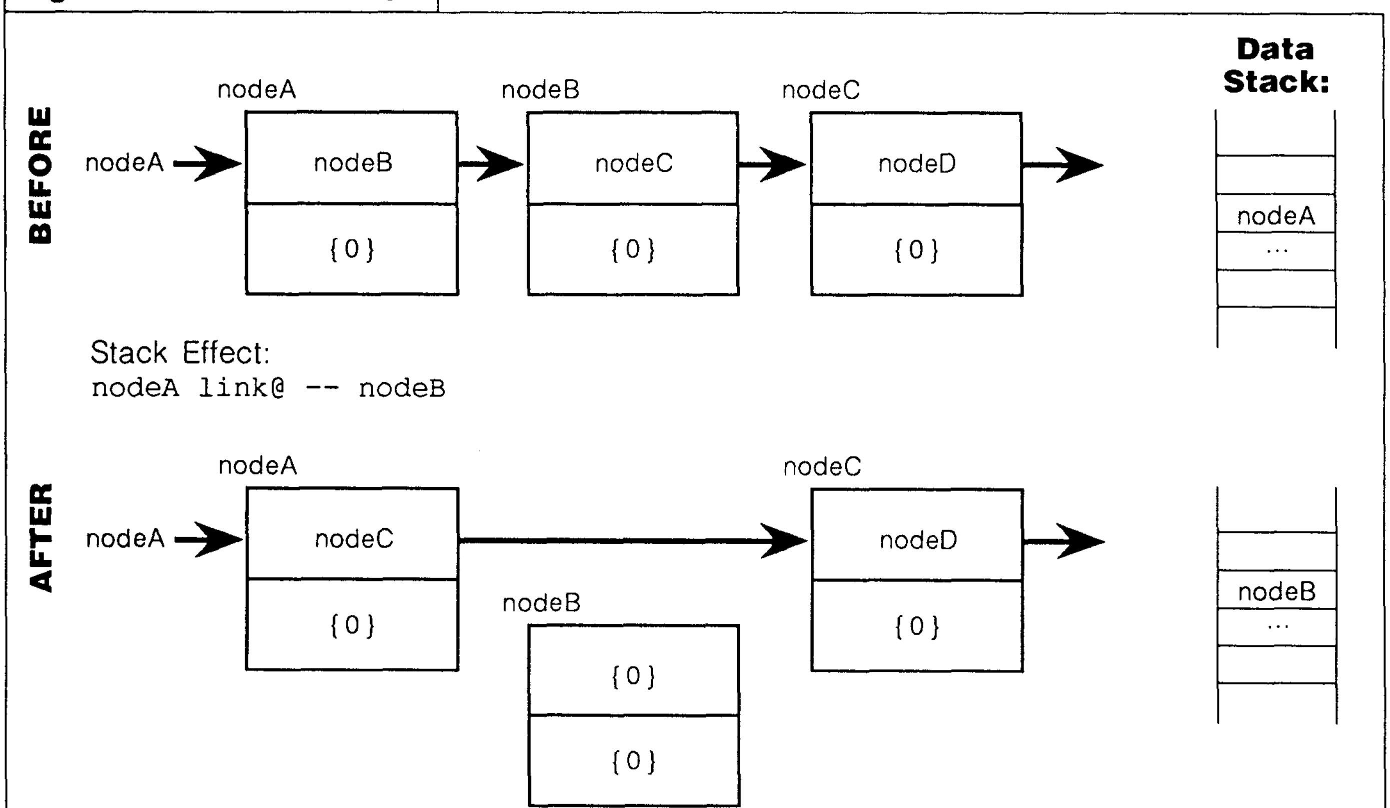
INTERNAL

The freepool is managed as a singly linked list which allows two operations: >free and free>. These names were chosen because their operation mimics the Forth return stack, accessed by >r and r>. In both instances, a single cell is pushed to or pulled from either the freepool or the return stack. The freepool need not be balanced in the sense of the return stack. Fetching a cell from the freepool, through free>, returns the address of a free two-cell node: either the first two-cell node from the freepool or, if it is empty, two cells allocated from the Forth dictionary. The freelist is kept intact when a node is removed. Nodes are returned to the freepool by >free. The address of the _free variable is used to indicate an empty list. Programmers using *modules* won't see _free, >free, or free> in dictionary searches. Therefore, the address of the _free value shouldn't be used anywhere outside the module. It's a better candidate for the empty sentinel than, say, zero. The cell pair managed by the freepool uses the first cell as the link field. There is no requirement for users of the freepool to use this approach. But links enforce this behavior.

A link is the single link from one node to its successor. Here, we use the first cell of a two-cell pair to hold the forward pointer. As discussed above, the word link, given a node, returns the node and the next node. Link@ and link! operate on links with the usual meaning of fetch and store. Given a node, link@ returns the next node while repairing the links around the returned node. In effect, it fetches the link. Similarly, link! takes a pair of nodes, storing the second as the link from the first. Link@ is used in _stack@, which is a further primitive in stack@ and queue@; link! is a primitive in stack! and queue!. Figures Four and Five show the effects of link@ and link!.

With _stack@, the underlying concepts start to come home in terms of being able to visualize the pictures through the words. Link cell+ @ puts the data on the stack, preserving the instance; swap saves the data, with the instance now on top; link@ plucks out the node which just yielded its data, and >free stores the node in the freepool. Set++ is a compromise made to keep all words as one-liners. (I like to use multi-line phrasing, but when I saw the opportunity to make the one-liner unanimous, I took it.) Set++ is set to advance the "last" pointer when items are added to a set instance. The leading number of @s is arbitrary and could be made random to give the set truly random behavior. The 2dup = if @ then adds a necessary "next" when the previous fetches have yielded the "last" node. What happens is, the node trio of "instance last first" is modified to "instance last {random}", the value is fetched (dup cell+ @) and stored in the "last" cell (rot cell+ !), which is otherwise empty, and the instance then points to random (swap !), which is the new last.

Figure Four. Effect of LINK@.



EXTERNAL

The word `empty?` uses `_free` as the sentinel for empty type instances, hiding the use of `_free`. Users see the value on the stack as a return value from fetches. Its only purpose is to indicate empty instances. The word `x-link`, pronounced "cross-link," exchanges instances of like types. Two pair of `link rot` put both links on the stack. A `rot !` reassigns one of the crosses, while `swap !` does the other.

The action is in the nine following words. Note that stacks are different from sets and queues. Set creation (`set:`) and fetching (`set@`) are identical to their queue analogs. These two types are implemented by rings. To be pedagogic, an intermediate type called `ring` should have held the queue definitions, with queue definitions using the ring types. So much for pedagogy. `Set!` uses the `queue!` with the added facility of moving the "last" pointer to randomize the set.

Stacks simply create a null single cell to hold the stack pointer. See Figure One for an empty stack. Stacking a value requires a pair of cells from the freepool. The value is stored (`tuck cell+ !`) and the linked list is restored (`swap link!`). Fetching is simple: non-empty stacks return the value from a stack fetch (`_stack@`), while empty stacks return the address of the freepool, again, only useful by comparison to `empty?`.

Queues (rings) and sets are created by allocating a single cell, pointing to a two-cell node, whose initial "next" pointer points to itself (see Figure One). Use of the freepool by the word `free>` either allocates two cells in the dictionary or a node from the non-empty freepool. Items are stored in the queue (ring) by `queue!`, where the value is stored by the `tuck @ cell+ !`, recalling the value is put in the "last" node and a new last node is linked on from the freepool. Discovering this order made it possible to insert without testing. `Free> tuck over @` produces the "new instance new last" nodes on the stack, and `link! !` re-establishes the links. Queues are fetched by constructing the two links "instance last first" and, if the last and first are the same, the queue is empty (`drop _free`), otherwise the value is fetched (`@ _stack@`).

Again, sets are identical to queues, except on storing, the "last" pointer is moved to simulate a random order of the set.

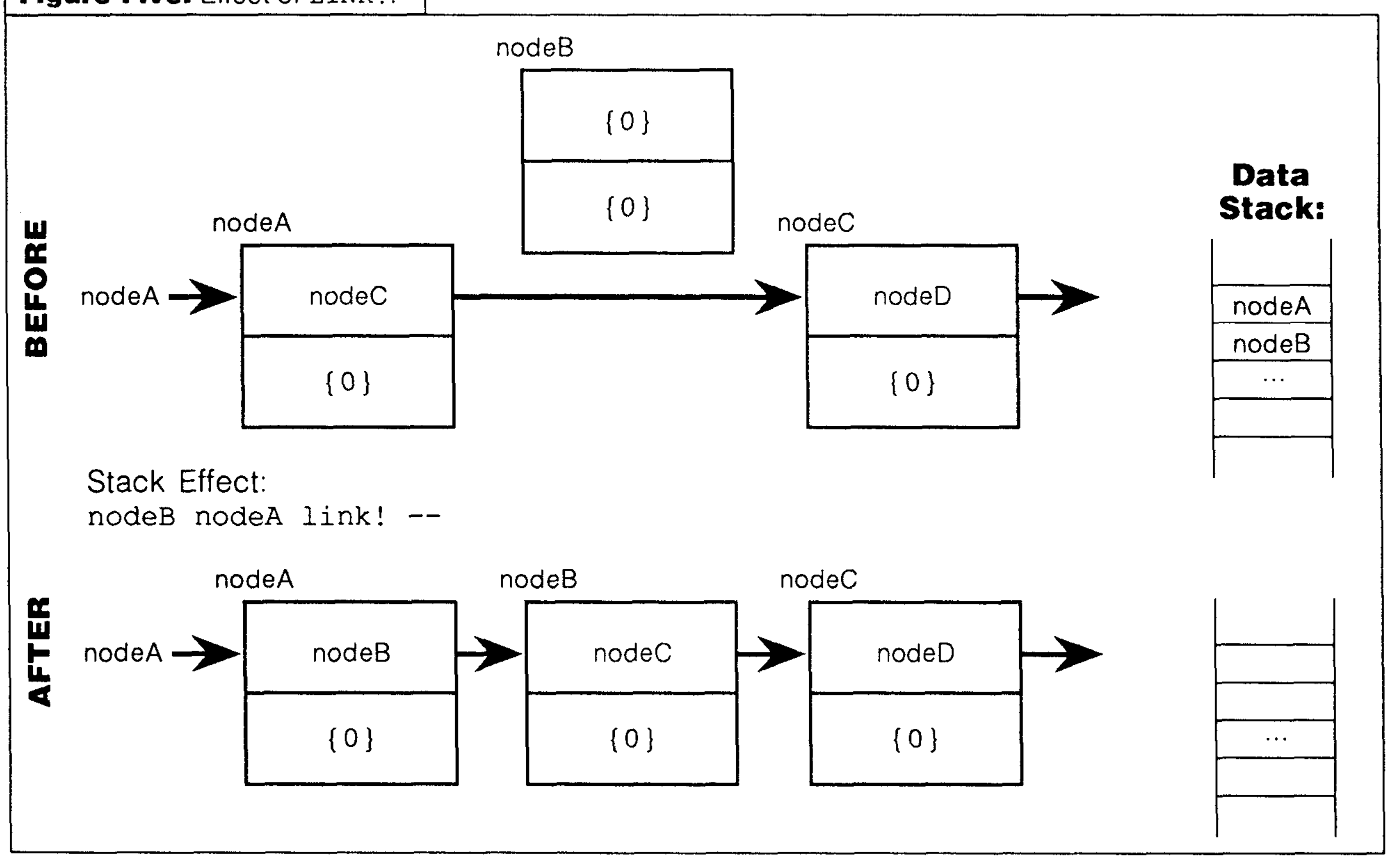
MODULE

Applications

A few simple applications in Listings Two and Three will serve to show some of the utility of the types. In a future article, we can examine how Wil Baden's recent macro-processor[2] might be done with queues. One of the interesting things I found in translating Wil's macros into queues is that he has discovered what I'll call a Text Multiplexor, or "TEX MUX" for short. It becomes more general in useful ways with queues. Queues introduce the

[2] "Macro Processing for Forth," Wil Baden, *Forth Dimensions*, Vol. XVII, No. 1, May-June 1995, p. 34–37.

Figure Five. Effect of `LINK!`.



Listing Two. Queue and set applications.

```
( eachmemb.fth -- for each queue member   --    950629 mcg )

  queue: empty-queue
: each-queue-member ( q xt -- )
   >r dup empty-queue x-link
   begin
      empty-queue queue@
      dup empty? 0= while
      dup r@ execute
      over queue!
   repeat 2drop r> drop
;
: dequeue ( q xt -- )
   >r begin dup queue@
            dup empty? 0= while
            r@ execute
   repeat 2drop r> 2drop
;
: do-set  ( set xt -- )
   >r begin dup set@
            dup empty? 0= while
            r@ execute
   repeat 2drop r> 2drop
;
            variable #count
: counter     drop 1 #count +! ;
: summer             #count +! ;
: clear-count      0 #count ! ;
: counted            #count @ ;
: .printer    cr 12 .r ;

             variable q_set
: q>set           q_set @ set! ;
: queue-set   q_set !    ['] q>set   each-queue-member ;
: queue-size clear-count ['] counter each-queue-member     counted
;
: queue-sum  clear-count ['] summer  each-queue-member     counted
;
: .queue     ." queue:" dup . space ['] . ;
: .set       ." set:"   dup . space ['] . ;
```

overkill of cell-sized elements, so where the data type is a character, queues might seem to waste space, but, as many things in Forth, it's small overhead for conceptual simplicity.

The few applications show how to navigate the data types. Two approaches are possible to visit each member in the type, either destructively or non-destructively. The default behavior is the "destructive" visit, where each member is removed when visited (see Listing Two). To visit and retain each member in the queue (in `each-queue-member`), we use an empty queue, exchange the pointers between empty and occupied queues, extract the successive elements from the temporary queue, execute a command on the extracted element, and restore the element to the original queue. For example, to count the members in a queue, use `queue-size`, which clears a counter (`clear-count`), ticks the counter for `each-queue-member`, and reports the `counted`. Queues are printed by `.queue`, which prints a leading message, prints the queue address (`dup .`), ticks the printer (`['] .`), and calls `each-queue-member` or `dequeue` to either print and preserve the queue or print the queue while emptying it as well (see Listing Three).

Future Directions

We could look at similar operations for the set and stack as we did for the queue. But rather than duplicate similar code (compare `dequeue` with `do-set` in Listing Two), I'll implement Wil Baden's macro processor using the queues, and then re-implement the two traversal operations as macros. A macro word `traversal:` creates an empty member type and defines two words (`each-`*`type`*`-member` and `do-`*`type`*). The two words are the destructive and non-destructive type traversals. With this word, all we have to do is declare:

```
traversal: set
traversal: stack
traversal: queue
```

to produce the code. The "easy" way to do the job is simply to copy the code from `each-queue-member` and `dequeue` for `each-set-member` and `do-set` and `each-stack-member` and `do-stack`. My only problem with this approach is, it violates a principal: my threshold of pain is three. Three what? Three of anything. In software, you might have two copies of similar code—a reader and writer, perhaps—and have captured all the necessary generality. But when you get to three similar instances, you can bet that, sooner or later, you will need four or more. It behooves us to generalize sooner than later. Forth, which encourages factoring, gives us the simple means.

As an exercise to the reader, think of creating the `ordered-list` type, based on queue and an execution token which returns the ordered sense of two-cell values, as follows:

```
' ordering-word
ordered-list: new-instance
```

where *`ordering-word`* has the stack effect:

```
: ordering-word
  ( v1 v2 --  -1 | 0 | 1 ) ;
```

and the return code tells whether or not v1 is less than, equal to, or greater than v2. The words `o-list!` and `o-list@` should behave as expected.

Marty McGowan (mcg@ustad.att.com) is a member of the technical staff at AT&T Bell Laboratories in Whippany, New Jersey. He uses software as a data-manipulation tool in the Wireless Communication Center of Excellence. He recently concluded an effort to re-assign (or "interleave") frequencies on the dispatcher base stations for a large eastern U.S. railroad. His wife, Pat, tests UnixWare OS at Novell. Their three children are at the School of Visual Arts, Moravian College, and in high school.

Listing Three. Using the applications.

```
\ non-ANS include, .r
include datatype.fth
include eachmemb.fth

  set: test-set
queue: test-queue
       test-queue
    8  over queue!
   13  over queue!
   44  over queue!
   -1  over queue!
  dup  over queue!
    1  over queue!

   dup queue-size 5 .r cr
   dup queue-sum  7 .r cr
   dup        .queue each-queue-member cr
   dup test-set queue-set
   dup        .queue dequeue cr
   dup queue-size 5 .r cr
   dup        .queue each-queue-member cr
  drop test-set .set do-set cr
  drop
```

Bounds Checking for Stacks

Adapted from comp.lang.forth

From: Russell Y. Webb

In a software system, what is the most efficient approach to checking stack under/over flow? Feel free to assume a stack implementation that optimizes bounds checking.

Some ideas I've thought of are:

1. Checking a bounding byte to make sure it hasn't been overwritten.
2. Calculating the actual bounding addresses and comparing them to the stack pointer (is there an efficient way to do this?).
3. Only checking the stack bounds every *n*th instruction.
4. Having the return and data stacks grow towards each other reduces the number of checks from four to three.

I'm interested in having a fairly secure, stack-based virtual machine, but it seems like a lot of overhead to check everything.

Any ideas are welcome.

From: Gordon Charlton

Off the top of my head, say each stack is max 1K bytes long, and start at 0400h and 0C00h (therefore ending at 07FF and 0FFF, respectively). Stack overflow or underflow in either stack will cause bit 10 of the appropriate stack pointer to change from one to zero, so AND them together and test bit 10.

If it is zero, you have a problem, so now it is time to figure out what went wrong and deal with it.

From: Elizabeth D. Rather

We check for underflow following complete execution of a word (i.e., when returning to the input source for further interpretation). This provides good feedback during development with negligible performance impact. Overflows are a lot less common, and are pretty easy to check for (and hard to miss, since the results are usually catastrophic). The exception is when a background task infrequently leaves a value; when this is suspected, it's easy to monitor it from another task.

We're pretty rigorous about testing for stack imbalance during development, and if you do this and check operator input for reasonableness, you'll have a pretty reliable system.

From: Roedy Green

If you have a segmented architecture, you can put the stacks in their own private segments. Then the hardware will not let you wander out of bounds.

You might also do it with paged hardware, by declaring a read-only page after the stack.

From: Dwight Elvey

Roedy's suggestion [above] only works if you have a machine that has some form of protected mode. Running a 32-bit Forth on a '386, '486, or '586, this is a good solution. But what does one do if they are running on a lesser µP?

If one was developing their own hardware, one would typically use some form of PAL, PLA, or GAL to do their address decoding with. It would be quite simple to extend this to include a simple hardware bounds check.

For those who are looking for a simple way to see where the stack has been after running some code, I have seen the trick of loading the memory with some simple pattern like 55AA, and then checking to see how things are later. This works surprisingly well. With this one, I have caught the occasional underflow that left the stack depth correct.

From: Anton Ertl

Use the MMU. Have a protected page before and after each stack. This can be done in many Unix systems with the mprotect or mmap system calls. Then the stack check is free and you get a segmentation violation signal upon overflow or underflow.

From: Marcel Hendrix

If something like this really is needed badly, you can load the SS register with a selector that has exactly the right segment limit. This is possible with protected mode Forths for the Intel '386 or better, when they use a threading model where the data stack is accessed with hardware stack instructions. I can see a possibility to protect three stacks in this way, using FS: GS: overrides (I consider this a software solution, but maybe you don't).

I like Gordon Charlton's ideas about stack checking a lot: make sure you crash violently whenever an error is made. The trick is to switch the data and the return stack pointers at random times. :-)

From: Gordon Charlton

[That] needs explaining, I suppose.

I wrote a slightly serious and mostly humorous piece called "Upside Down, Wrong Way Round, and Backwards" looking at three ways of turning Forth on its head, with some justification for each.

"Backwards" talked about writing code that would apparently run backwards (like Michael Gassanenko's system), to simplify coding an otherwise difficult set of problems, including pattern matching.

"Wrong Way Round" proposed a word to exchange the return stack and data stack pointers, thereby massively increasing the number of available return-stack-ops at a stroke. The justification for this ludicrous proposal was that the more fragile a system is, the sooner bugs will reveal themselves. (What would you prefer, a bug that crashes the system during development, or one that insidiously corrupts data two years after you installed it?)

"Upside Down" argued that ANS Forth would allow `CHAR+` to be defined as `1-` (and so on for `CHARS`, `CELL+`, and `CELLS`), which would be handy for testing programs for adherence to the standard, except that there is one standard word that screws it up.

From: Hans van der Vuurst

I added a stack checker to the Forth compiler; it counts the stack behaviour of each word and tells at compile time if the stack is bad. This helps speed up development time a lot—I don't debug until the compiler does not complain about bad stacks. I implemented this system in response to customers getting "stack overflow/underflow" messages while running the application after I had made little changes and was not able (willing) to check every single case of software execution. The compromise is to do less "dynamic" stack behaviour, such as pushing/popping elements in a loop. I swear by it...

From: Chris Jakeman

Russell Y. Webb writes:

> In a software system, what is the most efficient approach to checking stack under/over flow? Feel free to assume a stack implementation that optimizes bounds checking.

Stack Overflow:

The thorough (and expensive) way to check for stack overflow is to include checks in each primitive that adds value(s) to the stack, such as `DUP`, `OVER`, `SOURCE`, etc.; and for the return stack, `>R`, `:`, etc..

> Some ideas I've thought of are:
> 1. Checking a bounding byte to make sure it hasn't been overwritten.

Agreed. A less thorough, but cheaper, way is to add a margin above the stack space and fill this with recognisable values. Add a test to the interpreter loop within `QUIT`:

- If the last value in the margin has been changed, then serious overflow has taken place—advise the user to reboot.
- If the first value in the margin has been changed, but not the last, then warn the user that a non-fatal overflow has occurred. Also advise him how to increase the size of the stack!

Stack Underflow:

You can do something very similar for underflow.

Coding Support:

You could also consider tools which help the user to avoid writing code that misuses the stack, by comparing the stack depth at entry and exit of each word. Does the change in parameter stack match the stack comment? Has the return stack changed at all? (It shouldn't!)

These tools are helpful because they identify the faulty word as soon as it is executed. Of course, they are turned off after testing is complete. (Prof. Hoare describes this practice as throwing away your life jacket once your canoe reaches the open sea.

From: Michael L. Gassanenko

Russell Y Webb wrote:

> In a software system, what is the most efficient approach to checking stack under/over flow? Feel free to assume a stack implementation that optimizes bounds checking.

Okay, one more trick is based on [the fact] that the '386 and '486 do check bounds, even in real mode. If your stack bottom starts at address FFFFh (odd!), then stack underflow will cause an exception, and you will be able to see the register (if you use QUEMM or something like it); or hit reset, if you do not catch the exception number. :-)

The return stack will rarely underflow; at least, usually you will know that something bad has happened because the system will hang (in 99%, i.e., if you do not copy/restore the return stack).

> Some ideas I've thought of are
> 1. Checking a bounding byte to make sure it hasn't been overwritten.

A very useful approach: when I was debugging BacFORTH, my system used to report:

Stack Underflow
Stack Has Been Underflown
L-Stack Underflow
L-Stack Has Been Underflown

I added checks to INTERPRET, and used to add ?STACK in misbehaving definitions. The word R. that prints the trace of return addresses (using the R@ 2- @ >NAME .NAME principle) turned out to be very useful in ABORT diagnostics.

> 3. Only checking the stack bounds every *n*th instruction.

SP and RP are usually registers, the counter scarcely can be allocated in a register.

> 4. Having the return and data stacks grow towards each other reduces the number of checks from four to three.

Please, do not do that. There are words SP@, SP!, RP@, and RP!, and most people believe that stacks grow downwards.

From: Bruce McFarling

Michael L. Gassanenko wrote:

> The return stack will rarely underflow; at least, usually you will know that something bad has happened because the system will hang (in 99%, i.e., if you do not copy/restore the return stack).

If the return stack underflows from a runaway R>, you might pick that up on a parallel operand stack overflow (though only if the value is not consumed, so this is not ironclad). If it underflows through a misaligned R> right near the top of stack, a few dummy returns into a return stack underflow error report (logically) below the bottommost return into the interpreter would catch that. If it's effective enough, it would be efficient, since it adds overhead to the return stack initialization, rather than while running.

Since a runaway situation is likely to go into cybervoid, you might have the efficient (but not bulletproof) return stack underflow guard, along with a stringent check that is run when you have to debug a seriously misbehaving word.

Michael L. Gassanenko wrote:

> "4. Having the return and data stacks grow towards each other reduces the number of checks from four to three."
>
> Please, do not do that. There are words SP@, SP!, RP@, and RP!, and most people believe that stacks grow downwards.

This was the subject of a discussion a month or more ago, wasn't it? SP@, SP!, RP@, and RP! would seem to be pretty model specific; I say, if you want to optimize the model for stack checking, go ahead. (And with the above, it goes from one to two.)

Since return stack shenanigans are the least likely to be portable, and most likely to require re-writing for your specific model anyway; if you go with face-to-face stacks, let the operand stack grow down and the return stack grow up.

From: Paul Shirley

Michael L. Gassanenko writes:

> The return stack will rarely underflow, or at least usually you will know that something bad has happened because the system will hang (in 99%, i.e., if you do not copy/restore the return stack).

There's a hidden trap here. A stack bounds underflow will almost certainly crash the system; however, individual words popping too much return stack need not crash your program (I've seen code work 99% correctly whilst merrily dropping returns). By largely limiting the return stack to actual return addresses, Forth increases the chance that an underflow will simply cause the tail end of a routine to be skipped *without* any instantly fatal effects.

This tends to suggest to me that stack checking really should be done at a routine level.

From: Bruce McFarling

Of course, if the word has been exhaustively debugged in the interpreter, the return that would be skipped in the erroneous condition would be the return to the interpreter, so tucking a 'return stack underflow' return under the interpreter would help there. However, it would *only* help if the word has been well tested, and the test suite well-chosen; so, with St. Murphy at hand, his wonders to perform, checking for balanced return in process is probably worthwhile, especially when hunting a mystery bug (where, by definition, *some* of your exhaustive testing missed a trick somewhere).

From: Chris Jakeman

> Stack Overflow
>
> The thorough (and expensive) way to check for stack overflow is to include checks in each primitive that adds value(s) to the stack, such as DUP, OVER, SOURCE, etc. and for the return stack >R, :, etc..

Further to my previous post, I've been experimenting with a thorough way to check for data stack and parameter stack overflows.

Checks within the primitives (DUP, >R, etc.) detect overflow and execute -3 THROW or -5 THROW. CATCH and THROW are secondaries (defined in Figure One), so this is an unusual instance of a primitive executing a secondary! (Or, more precisely, arranging for a secondary to be executed next.)

But wait a moment. THROW will need some room on the data and return stacks to execute correctly. I handle this problem in the primitive checks. If they fail, they discard a few values from the appropriate stack before executing THROW.

I could avoid this by making THROW into a primitive which doesn't push anything onto the stacks. I don't want to do that because THROW calls a vectored word (i.e., 'UserThrow @ EXECUTE below) which supports some debugging. After THROW has been called, and before it restores the stacks to the depth saved by CATCH, the values on the stacks are precisely what is needed to find the cause of exception.

A debug word called at this point can present the data stack information as integers and the return stack information as a sequence of called words (or call tree).

It's an interesting paradox—THROW can call a debug word to show exactly what has gone wrong to cause the exception, but not after a stack overflow, because we have had to discard some values to allow room for THROW to operate!

Can anyone suggest a solution?

From: Bruce McFarling

Chris Jakeman wrote:

> But wait a moment. THROW will need some room on the data and return stacks to execute correctly. I handle this problem in the primitive checks. If they fail, they discard a few values from the appropriate stack before executing THROW.

Figure One. Jakeman's code for ANS CATCH and THROW (assumes RDepth similar to DEPTH).

```
VARIABLE CatchRDepth

: CATCH
  ( i*x xt -- j*x 0 | i*x n )
  DEPTH >R
  CatchRDepth @ >R
  RDepth CatchRDepth !
  EXECUTE
  R> CatchRDepth !
  R> DROP
  0
;

: RestoreDepth                      \ Adjusts the data stack to provide
  ( RequiredDepth -- )              \ the depth required not counting the
                                    \ parameter on top of the stack.
  >R DEPTH R>                       \ -- Actual Required
  2DUP > IF                         \ If Actual > Required ...
    DO  DROP  LOOP                  \ Drop surplus
  ELSE
    SWAP
    2DUP > IF                       \ If Required > Actual ...
      DO  0  LOOP                   \ Add zeroes
    ELSE
      2DROP
    THEN
  THEN
;

: RestoreRDepth                     \ Reduces the Return Stack to the
  ( RDepthRquired -- )              \ depth required.
  R>                                \ Save the next word.
  RDepth ROT -                      \ -- Actual-Required
  2DUP >= Assert
  BEGIN
  DUP 0> WHILE
    R> DROP
    1-
  REPEAT
  DROP                              \ the count
  >R                                \ Restore the next word
;

: THROW                             \ Taken from Standard
  ( k*x n -- k*x | i*x n )
  ?DUP IF
    'UserThrow @ EXECUTE ?DUP IF    \ This line not in Standard.
      CatchRDepth @ RestoreRDepth   \ Restore the Return Stack to depth
                                    \ saved by CATCH.
      R> CatchRDepth !              \ Restore value from any previous CATCH.
      R> SWAP
      >R RestoreDepth R>            \ Restore Data Stack as best we can.
    THEN
  THEN
;
```

Instead of discarding the information, store it in a private, dedicated stash location. If the word to do this is done as a primitive (appropriate, I believe, if there is a stack problem), it can avoid use of the stack.

From: Roedy Green

You could create a small emergency stack, and switch to it as part of calling THROW. THROW could then restore the stack (not its own, which makes life a little simpler), then switch the stack pointer back to point to the restored one. I do similar coding when I JAUNT in Abundance.

JAUNTing is a type of throwing where you restore past system state to give the illusion of running the program backward in time. It is used primarily for data entry, to let the user back up and change his mind about a previous decision keyed, or in response to failing an assertion.

From: Julian V. Noble

Stack underflow can be a problem, depending on whether the CPU generates exceptions or whatever. But anyway, checking for it on all operations that consume stack items can slow up a program. In my opinion, the best way to avoid underflow is to check each word as it is written, to make sure it does to the stack what is wanted, i.e., leaves it in the condition expected by the stack comment (which should be the minimum documentation accompanying any word being defined).

Stack overflow is easier. Overflow that crashes the machine happens only two ways: excessively deep recursion, or a loop containing a word that leaves too many things on the stack. The second is easy to avoid: one need merely factor out the contents of a loop as a word, and test that word for its stack effects *before* running the word with the loop.

Thus,

```
: inner   ( --)     stuff  ;

: outer   ( n --)   0 DO   inner  LOOP  ;
```

If you test inner before running outer, you can see immediately whether or not there will be trouble.

Recursion is harder. The trick here is to avoid algorithms that grow faster than log(N) with the problem size N. That is, recursion makes the return stack grow as the number of nested levels. On divide-and-conquer algorithms this will be log(N), which for many problems is tolerable without having to increase the size of data or return stacks. However, the Microsoft (!) example of string reversal (that is, abcdefg → gfedcba) is

```
function reverse$( s$ )
  c$ = left$( s$, 1 )
  if c$ = null$ then
    reverse$ = null$
  else
    reverse$ = reverse$( mid$( s$, 2 ) ) + c$
  end if
end function
```

which takes (N^2) time, and increases the depth of the data stack (and the return stack, if you were so foolish as to translate to Forth) as N^2 also. Guaranteed to crash on a long string. Don't use recursion to compute N! either.

Compound recursion applied to recursive-descent parsing should be safe, even if not entirely predictable, since the number of levels will increase only as log(expression length), for example.

From: Roedy Green

One way to check the return stack would be to salt it with five entries that point to a routine that complains and aborts. If somebody pops the real first element off the stack, then returns, it will hit one of these.

In practice, you will probably die long before that. If you mess up the return stack, it is because you did not match your >Rs and R>s. You will die long before you underflow or overflow the stack.

From: Claus Vogt

How about checking the return stack in each word? If each word began with a word which saves the return stack pointer and ended with a check for balance, you would not crash. For ease of use, the check may be globally enabled or disabled for following loaded words, by changing the behaviour of : and ;. (See source in Figure Two.)

But if we want to extend the error checking (maybe for educational purposes), other checks are necessary. Checking the balancing of the data stack inside loops would probably be the first candidate. And, even after eight years of Forth development, I sometimes change data and address for store operations (!)—not to talk about these horrible SWAPs in front of CMOVE.

Has someone invented a ProtectedForth which checks for such errors?

From: Jonah Thomas

Claus Vogt writes:

> Has someone invented a ProtectedForth which checks such errors?

My Stand4th checks those and a lot of others—it checks everything I could think of. The beta test version -.10 is on taygeta, and I'm slowly grabbing little chunks of time to put together version -.09.

I'd welcome feedback on it.

Figure Two. Vogt's method for checking the return stack.

```
\ Source for return-stack checking, not tested.  Claus Vogt 1995

\ not ANS-compatible:
\ ANS doesn't know rp@
\ ANS renames both compile and [compile] to postpone
\ trick with : : :  doesn't run on every Forth system

Variable oldrp  oldrp off                \ Saves rp between [rcheck and rcheck]

: [rcheck   ( ;r ret -- ;r oldrp ret )   \ Initialize R check
        r> rp@
        oldrp @ >r
        oldrp !
        >r ;

: rcheck]   ( ; r oldrp ret -- oldrp )   \ Ends R check
        r>
        r>
        oldrp @ rp@ - abort" R Stack not balaced"
        oldrp !
        >r ;


: test-err  [rcheck   r>   rcheck] ;    \ should abort on executing rcheck]

: test-ok   [rcheck                     \ prints out saved OLDRP and returnaddr
        r> dup . r> dup . >r  >r
        rcheck] ;

: : : compile [rcheck ;                 \ Not possible with every Forth system!
: ; compile rcheck]  [compile] ;  ; immediate

\ After redefinition of : and ; the following compiles exactly as test-err above

: test-err  r> ;   \ should abort on executing rcheck]
```

Nanocomputer Optimizing Target Compiler:

The Processor-Independent Core

Tim Hendtlass
Hawthorn, Victoria, Australia

This compiler shell has been written to assist programming modern nanocomputers, small single-chip processors with integrated RAM, ROM, and I/O. New nanocomputers appear regularly, and a simple alternative to assembly language can speed the development of applications. This shell provides a processor-independent core, described in this part, and only needs to be matched with a processor-specific library to provide a compiler that accepts Forth input and generates absolute machine code. In the second part, a library for the PIC16C71 and PIC16C84 processors will be presented. Using the description given here and that example, libraries for other processors can readily be developed.

The minimum processor-specific library is derived from the minimal set of primitive words in eForth. In eForth, all other words are derived from these primitives; these same derivations can be used here. You can, of course, define other words as primitives, in the interests of speed, but it is not required that you do so.

The compiler has been designed to support chips with different word lengths and different architectures; it only expects that the target processor executes a series of instructions taken from some type of ROM and has some RAM in which to keep variables and stacks.[1] Since it can support Harvard architecture processors (those with quite separate program and data spaces), as well as those based on the Von Neuman architecture, the control stack may be separate from the return stack. At this stage of development, only colon definitions, constants, variables, and literals are supported. Interrupt support is so processor specific that it has to be provided as part of a particular processor's library.The compiler takes as input a source written in Forth, and processes it in two passes through the source code (pass one and pass three). Between these passes, it carries out a special pass through the processor-specific library (pass two). During these passes, it places information into three separate regions. Figure One shows where the information is placed and where it comes from.

The three spaces are as follows. First, there is the library. This contains a number of definitions of standard Forth words and any special words written by the user that they wish to keep so they can be used in the future. When run, the definitions in the library cause some of the target processor's native code to be laid down in the image. Some extra (but temporary) definitions are added to the library during the first phase of the compilation. The image is where the final program is assembled ready to be downloaded into the target. All code in the image is written for the target processor and cannot (in general) be run by the host processor. As code is put into the image, a record of what has been loaded is kept in the target vocabulary along with special code that, when run, will add subroutine calls to the code being assembled in the image

In pass one, the source is read and checked against the words in the library. The number of colons in the source is counted—this will enable the final or top word in the source to be identified during pass three. As words are found, the count of how many times each will be used is updated. Any word not found in the library is ignored in this pass.

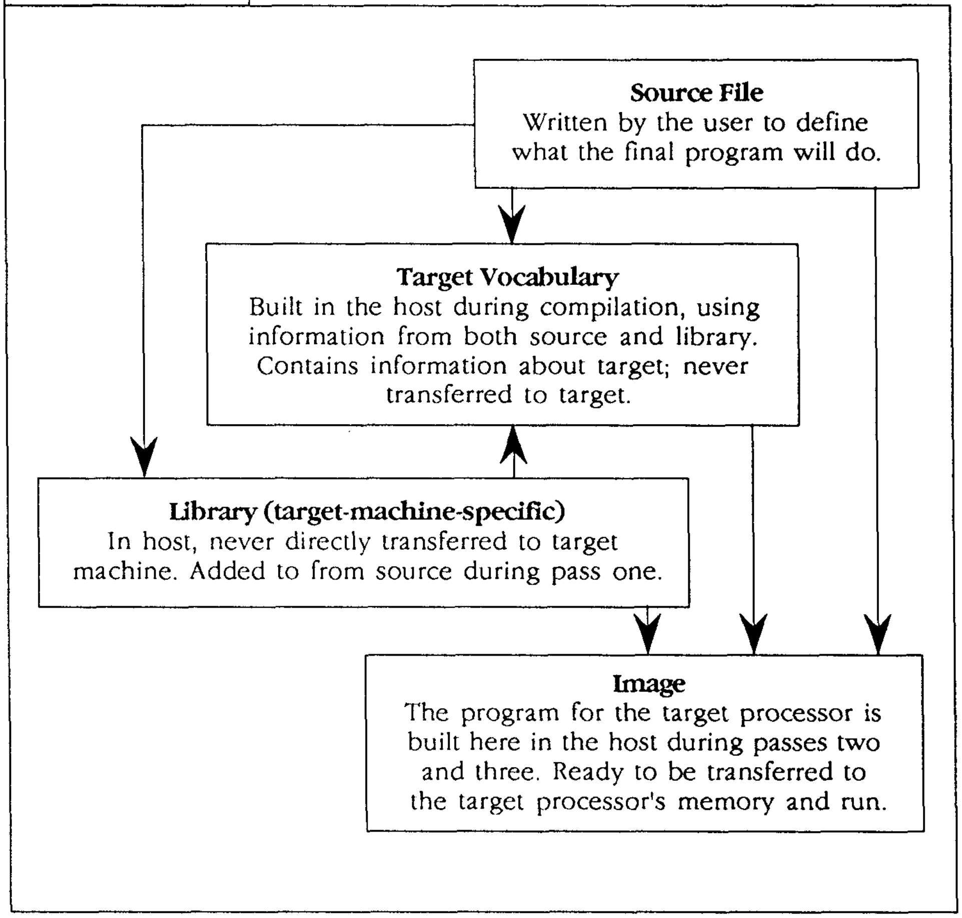
No code is laid down in the image during the first pass, but the library is added to. As constants, variable definitions, and literals (all of which will eventually cause a number to be put on the target processor's data stack) are encountered in the source, new (temporary) entries are added to the library. These will later be responsible for entering the code into the image which, when run in the target, will place the correct number on the target's stack. By the end of pass one, the library contains two types of entry: permanent library routines and transient numbers-handling routines. No matter the meaning of a number, as an address or a data value, a particular number value is only added once to the library.

For most entries, there are two ways they can be included in the final code. If they are used infrequently, it may be more economical on memory to just write their code in-line as and when needed. However, if they are used often, it will be more memory-efficient to load the code as a subroutine and call this as needed. For example, consider a routine for a PIC16Cxx that takes three words when written in-line but takes four words (the same three

[1] At the time of writing, this compiler has only been used to develop code for the PIC 16C71 and 16C84 processors. While care has been taken to try to make the core processor-independent, it is possible that some processor-dependence still remains.

words plus a return word) as a subroutine. Each time the subroutine is called, this takes another word. So, if this routine is used once, it makes more sense to write it in-line (three words) than to load it as a subroutine and then call it (four words in the subroutine and one in the call). However, if it is used twice, it would take six words in-line (three for each occurrence), and also six as a subroutine (four words in the subroutine and one for each call). Since there is no memory advantage either way, in this case it makes sense to load it in-line, as it will run faster in-line than as a called subroutine (each call and return takes time to execute). In this example, the break-even count is two; if it is used more than this, it is more memory efficient to load it as a subroutine. A subroutine that must, for some reason, be always loaded in-line (perhaps because it is a return stack modification word) can be accommodated by setting its break-even count to an absurdly high number. A routine that must always be loaded as a subroutine would be given a break-even count of zero.

Figure One.



During pass two, every library entry is checked to see if its use count (the number of times it will be used when the final code is built) will exceed the break-even count. If so, it is loaded as a subroutine. The actual code that loads it is not in the word `PASS2`, but in the `DOES>` section of the defining word `LIB:`. As each subroutine is loaded, an entry is also made in the target vocabulary so the compiler knows where this subroutine has been loaded in the image and can efficiently lay down a call to it in the image whenever it needs to. The loading of code into the image in pass two is a little bit more complicated than it at first seems. The reason already described in this paragraph why an entry may be loaded is the most obvious one (and is referred to in the source code as a *load-type one*). During this, a subroutine is constructed in the image and some of the words needed in this subroutine (let's call them subsidiary words) may themselves be words from the library. The loading of these words is referred to as a *load-type two*. If a particular subsidiary word has already been loaded as a subroutine, we just lay down a call to it. If it hasn't (pass one found that it would not be used enough to justify this), it needs to be written down in-line. So three types of additions may be made to the image in pass two: as a subroutine for later use, as a call to a subroutine that has already been laid down, or as a word laid down in in-line form.

In pass three, the source is read again and the image-code building is completed by adding all the user's colon definitions from the source file. As each word is extracted from the source, the pass three code will first check to see if an entry with the same name is already in the target vocabulary. All those words that were loaded as subroutines and earlier words defined in the user program will now be found in the target vocabulary. If a target entry is found, the compiler will lay down a call to the appropriate address in the image. If no target is found, the library code will be run which will lay down the in-line version of the code in the image. Every time a colon is encountered, the colon count is decremented; when the count reaches one, we are about to compile the top word, the word that runs the user's program. This definition is preceded by the initialization code needed (setting up the stacks, etc.). On power up, the processor executes the boot code, which jumps to this initialization code. After the initialization code is complete, execution falls through to the top word.

The final program we build in target space will have the structure shown in Figure Two. The address the target processor must jump to in order to start program execution is processor-dependent, and so is defined in the processor-dependent part of the library. Figure Two shows an example for the PIC16Cxx processors.

Assuming that the processor you wish to use has the capability to handle jumps and subroutine calls and some RAM in which to maintain stacks, this compiler can generate code for it. How the stacks are arranged and implemented is processor-dependent.

As an example, again for the PIC16Cxx processors, the normal processor return stack is used to hold return addresses, and the other two stacks (the data and control stacks) and the space for variables share RAM, as shown in Figure Three.

The compiler source is divided into two main files, the first of which includes the words that build library entries as well as the words that perform passes one, two, and three of the compilation, and all processor-independent library definitions. The second is the library file which has the few processor-dependent definitions. Each of these is loaded by, and on top of, F-PC. There could also be a file of convenience words that provide debug facilities, such as a copy of the image or a symbol table. The source of the program you wish to compile is written to a file and then compiled by typing `COMPILE <filename>`. After the compile has finished, the compiled code for the target is in the image space and can be extracted and loaded in EPROM or whatever is appropriate for your situation.

Figure Two.

Low program memory	Processor-dependent boot code (from library).
	Subroutines loaded from the library.
	First user colon definition compiled as a subroutine. May include in-line code from library and calls to the library subroutines loaded earlier in this target.
	Second user colon definition compiled as a subroutine. May include in-line code from library and calls to the library subroutines loaded earlier in this target.
	Third user colon definition compiled as a subroutine. May include in-line code from library and calls to the library subroutines loaded earlier in this target.
	etc.
	etc.
On start-up, processor jumps here to begin execution.	Initialisation code. Processor processes this and then falls through to the following code.
High program memory	Top user colon definition compiled. Must be an endless loop. Will include in-line code from library and calls to the library subroutine and other colon definitions already loaded.

The first compiling word (`Library-Routine`) is used to add library entries that define a routine for the final target processor to do. As with most compiling words, the objects that it produces have two parts—a private storage region for each word produced by the compiling word (each child word), and a pointer to the code that defines what the child word will do when it is run. All the children from one particular compiling word share the same run-time code.

Figure Three.

Low RAM	Control stack (dynamic size, grows down). ↓
	Spare space (may at times drop to zero).
	Data stack (dynamic size, grows up) ↑
High RAM	Variable storage space (fixed at compilation time).

The second compiling word (`Library-Number`) builds temporary library entries to handle numbers. These are similar, but simpler, structures to those produced by `Library-Routine`, but its children store different information in their structures. Again, while all `Library-Number`'s children have the same run-time code, this differs from the run-time code shared by all the children of the `Library-Entry` compiling word.

Each child of the `Library-Routine` entry compiling word has the structure shown in Figure Four..

The child also has a list (in F-PC's normal list space) that is the list of words that follow the name and precede the terminating semicolon. This list is pointed to by the entry at adr+7 and adr+8 in the child's private storage space.

For example, consider the processor-independent library entry for `NIP`.* This is defined as:

```
2 LIB: NIP swap drop ;
```

Two is the breakeven count for this word, assuming that each entry, including the return, occupies one word (if used more than this, load as a subroutine). Imagine that, when this definition is compiled in the host, the part of it in code space starts at 1000 hex and the part in F-PC's normal list space starts at 2000 hex. Also, suppose that the next library definition's code part starts at 1019 hex and

*This is just an illustration. Of course, `NIP` could be defined as a primary (machine language) word. In part two, the library for the 16C84, it is so defined as it is just one machine-code instruction.

Figure Four.

adr	adr+2	adr+4	adr+7	adr+9	adr+10	adr+10+n
# times this routine will be used.	breakeven count	jump nest	address of list for this definition	length byte of name of the entry (n)	ASCII name of entry	address of length byte of next library entry

Figure Five.

In code space:

1000 hex	1002 hex	1004 hex	1007 hex	1009 hex	100A hex	100D hex
0	2	E9 E0 88	2000 hex	3	"NIP"	1019 hex

In list space:

2000 hex	2002 hex	2004 hex
address of swap routine	address of drop routine	address of unnest routine

Figure Six.

adr	adr+2	adr+4	adr+6	adr+8	adr+9	adr+9+n
# times this routine will be used.	breakeven count	low 16 bits of number	high 16 bits of number	length byte of name of the entry (n)	ASCII name of entry	address of length byte of next library entry

that the routine NEST (the normal colon definition interpreter) is at 88E0 hex, so that `jump nest` in 80x86 machine code is E9 E0 88 hex. Then, the full entry compiled into the host will be as shown in Figure Five.

Each child of the library number entry compiling word has the structure shown in Figure Six, all of which is in the code space. There is no list associated with the library entry for a number.

When the child of either of these compiling words is run, it first puts the start address of its personal data area on the stack (indicated as adr above), and then jumps to executes the common run-time code for all children of this compiler. The code for either type of child is in three parts: first it checks what pass is currently being done, and then runs the code for that pass.

During passes one and three, a library word is found by looking up the name, almost as we would do with any Forth word. "Almost," as in pass one only the library vocabulary is searched, and any word not found there and which is not a number is ignored, rather than being considered an error. In pass three, both the target and the library vocabularies are searched, and if a word is not found in either, this is considered an error. When a library word is loaded into the image as a subroutine, an entry is made in the target vocabulary. Checking the target vocabulary first enables us to see if the word in question has already been added as a subroutine. If an entry exists (a subroutine has been loaded), it is run and lays down a call to the subroutine in the image. If no subroutine for this word has been loaded (there is no entry by this name in the target vocabulary), the library pass three code lays down the required word as in-line code.

During pass two, every library entry has to be checked to see if it is used enough to warrant loading as a subroutine. As the different words are distributed on different threads, it is not easy to ensure they are all checked in the correct order. It is mainly for this reason that the link field is added at the end of the private information area of each child word of either of the two compiling words. Each link field entry has the address of the count byte of the name field of the next entry. By following the chain, it is simple to access each definition in turn.

An example of a word (NIP) for the PIC16C71/84 series of chips was given above. NIP is a secondary word (it calls other Forth words). An example of a primary (one which calls no other library routine) is:

```
3 LIB: DUP dpt- pushdl ;
```

where *dpt-* is a machine-code word that lays down code to decrement the data-stack pointer (0384 hex), thus making the data stack, which grows down, one item larger; and *pushdl* is a machine-code word (80 hex) that copies the top of the data stack (in register W) to the address pointed to by the data-stack pointer (the new stack location we just acquired). The breakeven count of three ensures that DUP is loaded as a subroutine if it is used more than three times, or in-line if it is used three or less times. The code actually laid down if DUP is entered as a subroutine is 384 hex, 80 hex, 8 hex. For the PIC16Cxx processors 8 hex is the object code for return. If loaded in-line, 384 hex, 80 hex is laid down each time DUP is encountered in the source.

As well as words that will eventually cause code to be added to the image, the library also contains special versions of the standard Forth words :, ;, CONSTANT, and VARIABLE. These are run as these words are encountered in the source, and carry out the following actions.

The library colon compiler just counts the number of times it is called during pass one. In pass three, it first decrements this count and, if it is zero (the last colon definition in the source file is being compiled), runs INIT-CODE to lay down the initialization code needed. Then, no matter what the count is, it adds a new entry to the target vocabulary which consists of the name of this word (the next input word after the :) and the code which, when run, will lay down a jump to the position where the next word will be written to the image (which will be the first word of the colon definition itself).

The library semicolon compiler does nothing until pass three. Then, unless the count maintained by the colon compiler is zero, it terminates this word's definition with a return instruction. This may not actually involve adding any extra code. If the last instruction laid down was a call, this is changed to a jump, as the sequence *call xxx return* is functionally the same as *jump xxx* but the latter form takes less memory and runs faster. Of course, if the last instruction laid down was not a call, a return does have to be laid down. Using the example above, NIP would not require an explicit return, as the final word of its definition is DROP, which is always loaded as a subroutine. The in-line form of DUP, however, does not finish with a call, so an explicit return has to be added when it is loaded as a subroutine.

The library literal compiler checks to see if the number it wishes to compile already exists in the library (has been encountered before). If so, there is no need to add it again, but just to bump the use count of the one already there. If it does not yet exist in the library, it adds a library-number entry to the library (with the particular value stored in its private information area) and a use count of one. If a particular number is used often enough it, too, will be added as a subroutine and called as needed.

The library constant compiler uses the library literal compiler to first check if the value of the constant already exists in the library and adds it if not. It then lays down an entry in the target vocabulary which, when called, will just transfer control to the relevant number entry routine.

Finally, the library variable compiler first allocates space in the image for the variable and then, armed with this address, uses the library literal compiler to enter it in the library (unless it is already there). It then makes an entry in the target vocabulary which, when called, just transfers control to the relevant number-entry routine.

The control structures implemented in the processor-independent core—the *if else then*, *begin while repeat until again*, and the *for loop* groups—are defined using five processor-dependent words. One will lay down code to perform an unconditional jump (Tjump), one lays down code to perform a jump if the top of the target stack is true (Tjumpt), another lays down code to perform a jump if the top of the target stack is false (Tjumpf), and two move data between the control and data stacks. These last two are called by the traditional names >r and r>, although only if the control and return stacks are one and the same will these names be accurate. As normal in Forth, these words consume the stack items they test. Very limited checking is done using the same words that F-PC uses to ensure that the stack depth at the end of a control structure is the same as that at the start. If not, the structures are probably incorrectly constructed. It is quite possible to beat this checking so that an incorrect structure is accepted, but this compiler pays the programmer the normal Forth compliment: they are assumed to know what they are doing. The compiler will attempt to optimize the code, but will not try to second-guess what the programmer means. If you write an empty loop, it will be compiled, not omitted; presumably, you had some reason for writing it.

The optimization comes from loading in the most memory-efficient way, and from ensuring that numbers (be they literals, constants, or addresses) are only entered in the library once. A stub is also provided for processor-dependent peephole optimization with the variable *last-load-type*. For example, on the PIC16C84 it can take ten instructions to do a C@. If, however, you know that the last code laid down loaded a literal number # onto the data stack (two instructions), the code to load the number # can be converted into code to load the contents of the address #, which is also two instructions. Thus, you can save ten instructions. After code to load a number has been laid down, *last-load-type* is set to two (if a subroutine was called to do the job) or three (if in-line code was laid down). Normally it is set to one.

A few other words in the source are worthy of a brief note. *GET-LINE* acquires a valid line from the source, skipping empty lines and returning with either a line and a true flag, or a false flag if we have reached the end of the file. *NO-SEARCH* is a curious word whose only role is to undo a side effect of the standard colon compiler. The normal F-PC colon compiler (the one we use to compile NOTC itself) always takes the name of the vocabulary to which the definition is to be added and writes this over the top item on the context stack, the list of vocabularies to be searched to find words used in this colon definition. When we are compiling our special versions of words such as colon and semicolon, these special versions must not be used (they are only for use when `NOTC` is compiling a source), so the name of the vocabulary the special definitions are being put in must not be on the context stack. Since colon insists on putting it there, we have to use *`NO-SEARCH`* to remove it again before any damage can be done.

The final words to describe are *`IN-LIB?`* and *`IN-TARGET?`*. Each of these looks in one specific vocabulary to see if a word is there and, if it is, returns with its address. Because F-PC uses 64 threads within each vocabulary to speed searching, before we look for it we must first work out which thread the word would be on if it were in the vocabulary at all. We cannot use the normal word *`FIND`*, as it will search through all the vocabularies on the context stack and automatically abort if it can't find what it is looking for in any of them. We just need to know if a given word exists in a particular vocabulary, and will base our future actions depending on the result. Finally, `[LIB]` is used to force the library version of the following word to be run, even though the library is not in the current context-stack search path. It is used as *`[LIB] dup`* and is equivalent to writing:

`[ also library ] dup [ previous definitions ]`

Three special words are provided to assist this core in handling any processors. One, *`BOOT-CODE`*, is provided so that any special processor-specific initialisation can be done before any code is laid down. This could be loading a small core of words to set target hardware options, for example. The second, *`INIT-CODE`*, is provided so that one can load any code that must be loaded and executed by the target processor, such as initializing stack pointers, before the top word of the source is run. The last word, *`END-ROUTINE`*, is a routine to run at the end of compilation. This could extract the image and write it as a file in a format to suit a PROM programmer, for example. Or it could perform some final packaging pass on the image. For example, for the P21—which packs up to four five-bit instructions in each twenty-bit word, with some instructions being position-dependent—*`END-ROUTINE`* might perform the intelligent packaging pass required to produce final code.

The source code for the core, shown with this article, can be divided into five parts. First comes the part that defines the compiling words to build the library and the words to handle the passes through the user's source that build the image. Then comes the few processor-dependent words for the library (loaded from another file). Then starts the processor-independent part of the library which consists of secondary definitions built from the processor-dependent words and the processor-independent control words. Fourth comes the word that does it all, `COMPILE`. Finally come a few utility words that let you look at the image, see what library words have been used, and look at a symbol table.

The minimum set of processor-dependent words consists of only *`drop, dup, swap, over, !, @, c!, C@, r>, >r, 0<, and, or, xor,`* and *`um+`*. The processor-independent words are built from these. Secondary words can be taken from the source code for eForth, but no doubt everyone has words of their own devising that they use. These personal words go in the third section of the core source. If you define a machine code version of one of these secondary words in the processor-dependent file of words for some processor (for speed perhaps), you must then comment out the corresponding definition in the secondary word definitions. If you do not do this, you will end up using the secondary definition, not your hand-crafted, processor-dependent version.

Part two of this article has a sample processor-dependent set of words (somewhat richer than the minimum set) for the Microchip 16C71 and 16C84 chips. They should form a model for any other processor for which you wish to produce a compiler.

Code begins on next page...

Tim is a long-time Forth devotee. A professor in the School of Biophysical Sciences and Electrical Engineering at Swinburne University, he spends much of his time working with artificial neural networks and evolutionary algorithms in his capacity as Director of the Centre for Intelligent Systems. He escapes to Forth whenever he gets the chance, and has a dream of building a giant evolutionary algorithm computation array using Forth chips. He can be contacted at Swinburne (P.O. Box 218, Hawthorn 3122 Australia) or by e-mail (tim@bsee.swin.edu.au). The source code for this article can be obtained by anonymous ftp from brain.physics.swin.oz.au in pub/forth, as can electronic copies of his book on F-PC, *Real Time Forth*.

Author contemplating another day without Forth.

```
\ Nanocomputer optimizing target compiler shell. NOTC  Version 1.0
\ (Pronounced NOTCH) Processor independent core.
anew program
\                        ******** PART ONE *******

\ ******************************************************* SPECIAL VOCABULARIES
  vocabulary notc              \ one to hold compiler words
  vocabulary library           \ one for the library
  vocabulary target            \ one for image and target subroutine calls

\ ************************************ ADD DEFINITIONS TO OUR NOTC VOCABULARY
  only forth also notc also definitions

\ **************************** DEFERRED LINKS TO THE PROCESSOR DEPENDENT CODE
  DEFER ICALL                  \ convert the adr on the stack to a call and lay it down
  DEFER IRETURN                \ if last cell a call, make it jump, else add a return
  DEFER INLINE#                \ routine to load a literal in line in final target code

\ ****************************** VARIABLES, BUFFER, LIST AND ASSOCIATED WORDS
  VARIABLE PASS                \ which pass we are on
  VARIABLE LOAD-TYPE           \ primary or secondary type of load (see LIB:)
  VARIABLE LAST-LOAD-TYPE      \ may hold data to help peephole optimization
  VARIABLE #:                  \ number of : definitionss in source
  VARIABLE IPTR                \ points to start of last entry in target image
  CREATE WBUFF 34 allot        \ a small working buffer
  CREATE WBUFF1 34 allot       \ another small working buffer
  CREATE WBUFF2 15 allot       \ yet another small working buffer
: COPY-TO-WBUFF ( adr -- adr )
  dup wbuff over c@ 1+ cmove   \ copy string from adr to wbuff
  bl wbuff dup c@ + 1+ c!      \ add blank on end
;
: COPY-TO-WBUFF1&2 ( adr -- adr )
  wbuff wbuff1 over c@ 2+ cmove \ copy string with blank from wbuff to wbuff1
  copy-to-wbuff                \ new string into wbuff
;
\ We keep a simple linked list of words in the library (lib-list) for use in pass 2.
  VARIABLE LIST-END            \ points to zero address at end of list
: ADD here list-end @ ! ;      \ update last link address to point to the entry we are starting
: TO-LIB-LIST
  here list-end ! 0 ,          \ save address and place a zero address after name
;
  CREATE LIB-LIST -1 here add "  " to-lib-list +! \ build a list with empty zeroth entry
: POINT ( n - adr )            \ point to length byte of nth entry in list
  lib-list swap 0              \ set up loop
  do dup c@ 2+ + @ loop        \ loop down link addresses to nth address
;

\ ************************************************************ ERROR MESSAGES
: Error ( n -- ) cr ." FATAL ERROR!  "
   case 1 of ." Library list is corrupted!!" endof
   2 of here count type ."  is undefined!!" endof
   3 of ." Compiled program is too big!!" endof
  endcase cr abort
;

\ ************************************************************* UTILITY WORDS
: ?NEW-LINE
  #out @ 60 > if cr then       \ go to new line if past col 60
;
: .* ?new-line ." *"  ;        \ new line if at col 60 then print *
: .LENGTH                      \ show how much code produced so far
  ?new-line ."  Image length now " Iptr @ 1+ .
;
: GET-LINE  ( -- flag)         \ get a line of source, flag=0 if no more (end of file)
  begin lineread settib #tib @  \ get a line, 0=end of file
  0= if false true             \ end of file, exit leaving false flag
   else #tib @ 3 >             \ >3 means a usable line,
    if .* -2 #tib +! true true  \ show progress and ignore crlf or try again
    else false
    then
   then
  until
;
```

```
: GETWORD ( -- here )
  bl word ?uppercase            \ get next word from input to here, ensure in uppercase
;
: [LIB]                         \ compile library version of following word
  also library defined          \ add library and look up next word
  if X,                         \ if found compile it
  else 2 error                  \ if not report fatal error
  then previous                 \ remove library again
; immediate                     \ run [lib] as we compile
: NS
  previous also ; immediate     \ stops us searching current vocab while compiling the compiler
: P1? pass @ 1 = ;              \ we in pass 1?
: P3? pass @ 3 = ;              \ we in pass 3?

\ *************************************************** VOCABULARY ACCESS WORDS
\ Search vocabulary for a word, ptr points to string to search for, adr is where found
: (IN-LIB?)  ( ptr vocab-to-search -- adr true | ptr false )
  over swap >body hash @ (find) \ calc thread to look on and go look for it
;
: IN-LIB? ( ptr -- adr true | ptr false ) ['] library (in-lib?) ;
: IN-TARGET? ( ptr -- adr true | ptr false ) ['] target (in-lib?) ;
: FIND#? ( d# -- adr true | d# false ) \ look and see if # is already in library
  2dup (d.)                     \ convert to a string
  tuck wbuff2 1+ swap move      \ copy up to wbuff2
  wbuff2 c!                     \ put length in place
  wbuff2 dup c@ + 1+ bl swap c! \ add blank to end
  wbuff2 in-lib?                \ is it in the library already?
  if nip nip true
  else drop false
  then
;

\ ***************************** WORDS TO COMPILE TARGET ENTRIES INTO THE HOST
\ Build routine in target vocabulary using the name at adr1 which, when called, will load "CALL adr2" in
\ target space. Adr2 is the current address of Iptr+1 - the start address of word about to be laid down
: BUILD_TVOC_ENTRY ( adr1 -- )
  also target definitions
  "CREATE Iptr @ 1+ ,           \ create header, lay down address that will be called
  previous definitions
  DOES> @ Icall                 \ lay down a call to stored number
;
\ Compile library routines, eg  n LIB: fred ; , where n is the breakeven count
: LIB: also library definitions \ add this definition to the library
  getword copy-to-wbuff         \ get name to use to wbuff
  drop wbuff "create            \ build header
  0 , , 233 c, >nest here 2+ - , \ use counter (0), breakeven count, install jump nest
  xhere paragraph + dup xdpseg ! \ paragraph align end of list space
  xseg @ - , xdp off !csp ]     \ enter list adr, 0 length, build liststop at ;
  add wbuff here over c@ 2+ cmove \ add name to library list (include blank on end)
  here c@ 2+ dp +! to-lib-list  \ move pointer to enclose name and complete the list entry
  previous definitions          \ back to adding to notc
  DOES> >r pass @ case          \ save pointer to entries info on return stack
   1 of 1 r@ +!                 \ bump usage count
     r@ @ r@ 2 + @ <=           \ use still below breakeven count?
     if r@ 4 + execute then     \ yes, run definition to see what it uses
   endof
\ Pass 2 code may be run as we load a heavily used word as a subroutine (type1)
\ OR as a heavily used word we are loading in turn uses this word (type2).
\ In type 1, since no word can be called by an earlier one, we cannot be in
\ target and will be loaded as a subroutine if our own use is high enough.  In
\ type 2 if we have already been loaded as our use is high and now a later
\ word needs us, lay down a call to ourself in the target.  If not already
\ loaded just write ourselves in line.
   2 of r@ 9 + in-target?       \ we already exist in the target?
     if execute                 \ yes type2, load call to us
     else drop r@ @ r@ 2 + @ >  \ no, is actual use > breakeven count?
      if r@ 9 + build_tvoc_entry \ yes type1, add entry to target vocabulary
       2 load-type !            \ show now doing type two as load this word
       r@ 4 + execute Ireturn   \ load in line and convert to subroutine
      else load-type @ 2 =      \ In type2 ONLY we should now load inline
       if r@ 4 + execute then   \ It is type2, load in line
      then
     then
```

```
   endof
   3 of r@ 9 + in-target? not   \ point to ASCII name, not already loaded as subroutine?
    if drop r@ 4 + then execute \ yes load it in line, no use subroutine
    1 last-load-type !          \ optimization flag
   endof
  endcase r>drop                \ clean up return stack
;
\ Add a # to library unless it is already there when we bump it's use count.
\ Expects text version of # in wbuff.
: ADD#-TO-LIBRARY ( # -- )
  also library definitions      \ where we need to add it
  wbuff "CREATE                 \ name from wbuff for library entry
  1 ,  2 , swap , ,             \ use (init to this 1), breakeven count, 32 bit value low, high
  add wbuff here over c@ 2+ cmove \ add name to library list
  here c@ 2+ dp +! to-lib-list  \ move pointer to enclose name and finish the list entry
  previous definitions
  DOES> >r pass @ case          \ save pointer to entries info on return stack
   1 of 1 r@ +! endof           \ bump our usage count
   2 of r@ @ r@ 2 + @ >         \ actual use greater than breakeven count?
    if r@ 8 + build_tvoc_entry  \ add an entry to the target vocabulary
     r@ 4 + 2@ inline# Ireturn  \ load as a subroutine
    then
   endof
   3 of r@ 8 + in-target?       \ already loaded as a subroutine?
    if execute                  \ yes run that
     2 last-load-type !         \ optimization information
    else drop
     r@ 4 + 2@ inline#
     3 last-load-type !         \ optimization information
    then
   endof
  endcase r>drop                \ lose pointer
;
: LIBRARY-NUMBER        ( d -- )
  wbuff in-lib?                 \ already in the library? (we had this # before?)
  if execute 2drop
  else drop add#-to-library     \ if so run it to bump its count
  then                          \ no, go add it
;
: ADD-CONSTANT-TO-LIBRARY
  also library definitions
  getword copy-to-wbuff         \ get next word in input stream
  drop wbuff "create            \ build header from it
  wbuff1 in-lib? not            \ find # just entered (text in wbuff1)
  if wbuff1 2 error then        \ disaster if not found
  -1 over >body +!              \ adjust count (this isn't a real use)
  ,                             \ store address of run time code for number
  previous definitions
  does> @ execute               \ go do this routine whenever constant name is used
;
: POINT-TO-NUMBER ( adr -- )    \ build entry pointing to code for a number
  also library definitions
  wbuff "create ,               \ build header, use adr of run time code for #
  previous definitions
  does> @ execute               \ at run time just run the number
;

\ ************************************************HIGH LEVEL COMPILING WORDS
: INITIALIZE                    \ ensure that everything is clean before we start.
;
: PASS1                         \ READ THE SOURCE FILE PERFORMING ACTION ON EACH WORD
  1 pass ! begin get-line       \ try for another line to process
  while                         \ one is available
   begin getword c@ 0<>         \ get word, is one available?
   while here copy-to-wbuff1&2  \ if so, save word we are working on
    number?                     \ is it a number?
    if library-number           \ yes, add to library UNLESS already there!!
    else 2drop wbuff in-lib?    \ if not, is it a library word?
     if execute else drop then  \ yes run it, no ignore it
    then
   repeat
  repeat                        \ go load a new line
;
```

```
: PASS2                           \ load all frequently used library words as subroutines
  2 pass ! lib-list 2+ @          \ point to first real entry
  begin .* dup in-lib?            \ show progress,find where it is in library
   1 load-type !                  \ mark as a primary load (type 1) - LIB:
   if execute else 1 error then \ if found run it, fatal error if not present in library
   dup c@ + 2+ @ dup 0=           \ get next address, check if 0 (end of list)
  until drop                      \ continue until it is, then lose the zero
;
: PASS3                           \ Search target then library ONLY to lay down the code
  3 pass ! begin get-line         \ get a line to process
  while begin getword c@ 0<>      \ get word
   while here in-target?          \ got one, in target?
    if execute else in-lib?       \ yes, run it, no check library
     if execute                   \ yes run it
     else 2 error                 \ fatal error if not found
     then
    then
   repeat                         \ go get next word
  repeat                          \ go load a new line
;
WARNING OFF                       \ we will redefe all sort of things deliberately!

\                       ******** PART TWO *******

\ ***************************************************************************************
 FLOAD PIC84LIB.SEQ               \ load the processor dependent library
\ ***************************************************************************************

\                      ******** PART THREE *******

\ ****************************** THE PROCESSOR INDEPENDENT PART OF THE LIBRARY

  only forth also notc also       \ search notc>forth>root
  library definitions             \ add to library
  previous also                   \ but remove library from search list

\ Library needs patches to regular words \ and (   so all comment defining words work when we are only
\ searching the library. Patch entry technique works because latest definiton is hidden until complete.

: \ [compile] \ ; immediate
: ( [compile] ( ; immediate

\ Special versions of : ; CONSTANT VARIABLE IF ELSE THEN BEGIN UNTIL AGAIN WHILE REPEAT
\ Don't search the library as we load them or we will try to use these versions as we compile!

: :
  ns pass @ case
   1 of 1 #: +! endof             \ increment count of # colons in source
   3 of -1 #: +! #: @ 0 =         \ decrement count, this the last : definition?
   if [lib] init-code then        \ load initial code if so
    getword build_tvoc_entry      \ get name for new routine and build a header
   endof drop
  endcase
;
: ;
  ns p3? if                       \ if not pass 3
   #: @ 0 <> if                   \ and not last word
     Ireturn                      \ add return or make last call a jump
   then
  then
;
: CONSTANT
   ns pass @ case
   1 of                           \ add constant name to library pointing to last number
     add-constant-to-library
    endof
   3 of                           \ lose the number we added and skip over name
     [lib] remove# getword drop
    endof
   endcase
;
: VARIABLE
   ns pass @ case
```

```
  1 of [lib] var-space          \ allocate space for a variable
   getword copy-to-wbuff drop   \ get name to use to wbuff
   find#?                       \ look for this number in the library
   if point-to-number           \
   else add#-to-library         \ add entry for this variable to library
   then
  endof
  3 of getword drop endof       \ just skip name in pass 3
 endcase
;
\ ********************************************** PROGRAM FLOW CONTROL WORDS
: IF ( -- adr ) ns p3?          \ only any action in pass 3
 if !csp Iptr @ 0
 [lib] Ijumpt 1 then            \ get current adr, build jump 0, show address from if clause
;
: THEN ( adr flag -- ) ns p3?   \ only any action in pass 3
 if Iptr @ >r swap Iptr !       \ save current address, go back to dummy jump we laid down
  1 = if r@ [lib] Ijumpt
      else r@ [lib] Ijump       \ lay it dowm again, this time with the correct address
      then
  r> Iptr ! ?csp                \ back to where we were, check control structure
 then
;
: ELSE ( adr1 flag1 -- adr2 flag2 )
 ns p3?                         \ only any action in pass 3
 if drop Iptr @ 0 [lib] Ijump   \ and lay down dummy unconditional jump, save its address
  swap Iptr @ swap Iptr !       \ go back to rebuild the dummy jump with correct address
  dup 1+ [lib] Ijumpt Iptr ! 2  \ build it and come back, show address is from an else
 then
;
: BEGIN ( -- adr )
 ns p3? if !csp Iptr @ then     \ in pass 3 get address to branch back to
;
: UNTIL ( adr -- )
 ns p3?
 if 1+ [lib] Ijumpt ?csp then   \ in pass 3 lay down conditional jump check for error
;
: AGAIN ( adr -- )
 ns p3?
 if 1+ [lib] Ijump ?csp then    \ in pass 3 lay dowm unconditional jump check for error
;
: WHILE ( adr1 -- adr1 adr2 )   \ adr1 adr of begin, adr2 adr of while jumpf
 ns p3?
 if Iptr @ 0 [lib] Ijumpf then  \ in pass 3 lay dowm dummy jump if false, record address
;
: REPEAT ( adr1 adr2 -- )
 ns p3?
 if swap 1+ [lib] Ijump Iptr @  \ build unconditional jump back to begin, save current address
  swap Iptr ! dup 1+            \ save current address
  [lib] Ijumpf Iptr ! ?csp      \ resolve jumps check for errors
 then
;
comment: *******************REST OF PROCESSOR INDEPENDENT PART OF THE LIBRARY
 Now the extra library words from Eforth or elsewhere.  They are added to the
 library and use the words from the library.  If you need to use the regular
 forth words IF, AND, OR etc, these will need to be preceeded with [ also
 forth ] and followed by [ previous ] like in the words above (which needed
 definitions from the library which was not generally in the search path).
 They are entered into the library with LIB: - their breakeven count will
 probably be 1 and could be processor dependent.
comment;
only forth also notc also library also definitions
\ ************************************************************************
\ library entries go here
\ ************************************************************************

\                        ******** PART FOUR *******

\ ***************************************************THE WORD THAT DOES IT ALL
\ Use as COMPILE FRED.SEQ
only forth also notc also definitions
: COMPILE
  sequp file [lib] boot-code    \ open file, do any processor specific initialization
```

```
  cr ." pass 1 " pass1 .length  \ do pass 1 from start of file
  cr ." pass 2 " pass2 .length
  0.0 seek                      \ back to satrt of source file
  cr ." pass 3 " pass3 .length  \ do pass 3
  end-routine seqdown cr        \ do finish up and close file
  ." Final image size = "
  iptr @ 1+ . ."  words " cr    \ report on the final size
;

\                       ******** PART FIVE *******

\ ************************************************************** DEBUG WORDS
\ Image display words.
: SHOW_LIB_ENTRY ( n --  adr1 ) \ adr1 = address of next entry
  point dup>r in-lib?           \ get name and position in library
  if >body @ dup 0 <>           \ is it used?
   if r@ cr count type          \ point to name and type it
    30 #out @ - 0
    ?do ." ." loop              \ write dots to column 30
    ." used " . ." times"       \ show how many times used
    ."  loaded "
    r@ in-target?
    if ." as subroutine at "
     >body @ .                  \ show where loaded
    else drop ." inline "       \ or if loaded in line
    then
   else drop                    \ clean up pointers
   then
  else 1 error
  then r> count + 1+ @          \ calculate adr1
 ;
 : .LIB
  cr ." Library usage"
  1 begin                       \ start with the first
   dup show_lib_entry 0 <>      \ show one
  while
   1+                           \ as long as not at end, move onto next
  repeat drop cr
 ;
 : .SYMBOLS                     \ show all the user defined words
  cr ." Symbol table " cr
  ['] target >body here 500 +
  #threads 2* cmove             \ copy threads up in memory as we will alter them
  begin here 500 + #threads
   largest dup ?keypause        \ in case we want to see a big list on the screen
  while dup l>name dup w.id     \ print name
   40 #out @ - 0 do ." ." loop  \ write dots to column 40
   name> >body @ dup .          \ write address in decimal
   [compile] hex                \ switch to hex
   ." [" 4 u.r ." ]"            \ write address again
   [compile] decimal cr         \ revert to decimal, new line
   Y@ swap !                    \ ready for next entry
  repeat 2drop
 ;
 : .IMAGE
  cr ." Memory Map"
  cr ."  Address        Contents" cr
  iptr @ dup 1+ 0 do            \ set up loop
   #out @ 0 =
   if [compile] decimal         \ back to decimal
    i 4 u.r                     \ address in decimal
    [compile] hex               \ to hex
    ." .[" i 3 u.r ." ]"        \ address in hex too
    4 0 do ." ." loop           \ write dots
   then i iptr ! [lib] i@       \ set up pointer and read contents
   ." [" 4 u.r ." ]"            \ write in hex
   #out @ 60 >                  \ past column 60?
   if cr then                   \ start new line if so
  loop iptr !                   \ restore original tpointer
  [compile] decimal             \ final go back to decimal

 ;
 : PRINT-OUT printing on .lib .symbols .image printing off ;
```

Forth On-line

About half these entries are resource-provider responses to our survey, easily identifiable by the rich lode of information they offer. Sparser entries were derived from a quick login and browse simply to verify the presence of Forth. It is not our role to interpret the intentions or to verify the claims of resource providers. No doubt, there are some omissions and errors; apologies for those in advance—please bring them to FORL's attention by sending e-mail to forl@artopro.mlnet.com. (FORL is an electronic mailbox for tracking publicly available, Forth-related electronic resources; it is provided and maintained by Kenneth O'Heskin.)

Guide to Line Numbers
1.0 .... Resource name
1.1 Resource startup date
2.0 .... Location
3.0 .... On-line address/telephone numbers
4.0 .... Sponsorship
4.1 Sponsoring person/institution's name
5.0 .... Contact name (admin, sysop, etc.)
5.1 E-mail address
6.0 .... Access type (free/pay, conditions of access)
7.0 .... Connection type (modem/telnet)
7.1 Modem (maximum bps, parity/bits/stop)
7.2 Telnet (address)
8.0 .... Approximate number of Forth-related files
8.1 Theme of these files
8.2 Available to first-time callers?
9.0 .... Mail and news
9.1.0 Mail technology
9.1.1 Binary mail tranfers supported?
10.0 .. System software, if relevant
11.0 .. Additional comments

Bulletin Board Systems

1.0 *Arcane Incantations*
1.1 Mar. 93
3.0 617-899-6672
5.0 Gary Chanson
5.1 gary.chanson@channel1.com
8.0 Several files (some authored by sysop), first-time caller available.
10.0 PC Board

1.0 *Art of Programming BBS*
1.1 Jan. 91
2.0 Mission, BC, Canada
3.0 604-826-9663
4.0 non-profit
4.1 ForthBC Computer Language Society
5.0 Kenneth O'Heskin
5.1 koh@artopro.mlnet.com
6.0 Free dial-up access for all Forth files.
7.0 modem
7.1 v32 8,N,1
8.0 hundreds
8.2 first-time callers ok
9.0 Mail and news; e-mail by low-cost annual subscription; Usenet groups (incl. comp.lang.forth); BCbbs.net.
9.1.0 uucp, qwk
9.1.1 uuencode/decode
10.0 WildcatJGNT_Mail
11.0 Download aop.zip for a list of all files on the board.

1.0 *The FROG Pond BBS*
1.1 Aug. 89
2.0 Rochester, NY, USA
3.0 716-461-1924
4.0 non-profit
4.1 The FROG Computer Society
5.0 Nick Francesco
5.1 nickf@vivanet.com
6.0 free
7.0 Modem
7.1 14400 8N1
8.0 5
8.1 languages
8.2 yes
9.0 Fidonet and Internet mail available for all users.
9.1.0 qwk,netmail
9.1.1 uuenc/decode
10.0 Remote Access (for now)
11.0 Download FROGPOND.EXE for self-extracting list of all files. All Forth files available to first-time downloaders.

1.0 *Gold Country Forth BBS*
2.0 CA, USA
3.0 916-652-7117
5.0 Al Mitchell
8.1 Some product support (password required), many free files.
8.2 Okay for first-time callers.

1.0 *LMI Forth BBS*
1.1 Oct. 84
2.0 Los Angeles, CA, USA
3.0 310-306-3530
4.0 business
4.1 Laboratory Microsystems Inc. (LMI)
5.0 Ray Duncan
5.1 sysop@lmi.la.ca.us
6.0 free
7.0 modem
7.1 1,200 – 28,800 baud, 8/N/1
8.0 hundreds
8.1 Mostly compatible with LMI Forth products, but also some public-domain Forth stuff.
8.2 yes (except for LMI product updates, which require prior registration)
9.0 Supports Internet e-mail and UseNet News
9.1.0 UUCP
10.0 PC Board 15.2
11.0 The LMI Forth BBS is primarily intended for technical support of LMI customers. However, all members of the Forth community are welcome to upload/download files in the public directories, and to use the LMI BBS for Internet e-mail and reading the UseNet comp.lang.forth conference.

1.0 *MindLink!*
2.0 Vancouver, BC, Canada
3.0 modem: 604-528-3500 (main) 28.8Kbps
Telnet: mindlink.bc.ca
4.0 Business
6.0 Pay; may log on as guest.
7.0 28.8Kbps, Telnet
8.0 75
8.0 Available only to registered users.
11.0 Two Forth file libraries: Sources.Forth and MsDos.Forth.

1.0 RCFB "The Rocky Coast Free Board"
1.1 Oct. 88
2.0 Golden, CO, USA
3.0 303-278-0364
4.0 private
4.1 Jax
5.0 SYSOP
5.1 jax@well.com
6.0 Free, but must register.
7.1 19200, 8-n-1
8.0 300
8.1 Programming tools and productivity
8.2 Must register online, wait 24 hours.
10.0 PC Board since 1988, Linux by mid-1996.

FTP Sites

1.0 Asterix Forth archive
2.0 Portugal
3.0 asterix.inescn.pt /pub/forth
4.0 university
4.1 Computer Graphics and CAD group INESC
5.1 paf@porto.inescn.pt
6.0 anonymous ftp
8.0 hundreds
11.0 First internet site of the GEnie Forth archives, built with the assistance of Doug Phillip's FNEAS server. Mirrored on hp.com.

1.0 Cygnus Support Ftp Service
3.0 ftp://ftp.cygnus.com
http://www.cygnus.com
5.1 info@cygnus.com (?)
11.0 This site has a good file list and appears to support some Forth material not available elsewhere on the net.

1.0 Faré's own small FTP site, Forth subsection
1.1 1994
2.0 Paris, France
3.0 ftp://frmap711.mathp7.jussieu.fr/pub/scratch/rideau/
5.0 François-René "Faré" Rideau
5.1 rideau@ens.fr
6.0 free (anonymous FTP)
8.0 Two FORTH systems, my port of eForth to Linux, and Olivier Singla's FROTH.
8.2 yes
10.0 SunOS4.1.3
11.0 This site does not contain much about Forth, but more is welcome if you upload it. I am developing my own system, TUNES, which is remotely Forth-related, and for which I opened this site.

1.0 Hewlett-Packard
3.0 ftp://col.hp.com/mirrors/Forth
6.0 anonymous ftp
11.0 Mirror site for asterix, recommended for North American users when asterix is busy.

1.0 i/tForth-specific stuff
1.1 Sept. 94
2.0 Eindhoven, Brabant, the Netherlands
3.0 ftp iaehv.iaehv.nl, directory pub/users/mhx
4.0 private
4.1 Marcel Hendrix
5.0 Marcel Hendrix
5.1 mhx@iaehv.iaehv.nl
6.0 free, anonymous ftp
8.0 10 – 20
8.1 i/tForth specific files, not ANS enough to put them on taygeta or such. Some very Intel-hardware-specific (networking, audio CD). i/tForth general info, release notes, previews.
11.0 There is a link on taygeta to this directory.

1.0 SimTel
3.0 ftp://ftp.coast.net/SimTel/msdos/forth
5.1 service@coast.NET
11.0 Several Forth files; and Norm Smith's Until revisions are updated here.

1.1 July 95
2.0 Ann Arbor, MI, USA
3.0 ftp://williams.physics.lsa.umich.edu/pub/forth
4.0 university
4.1 Particle Theory Group, Physics Department, University of Michigan
5.0 David N. Williams, sysadmin
5.1 David.N.Williams@umich.edu
6.0 free, low traffic, download only
7.0 anonymous ftp
8.0 12–20
8.1 Forth: personal interests of David N. Williams
11.0 This is one directory at an anonymous FTP site devoted mainly to communication between our group and the particle theory community. Forth and symbolic computing (Schoonschip) happen to be an interest of one of our group.

FTP/Web Sites

1.0 Forth Research at Institut fr Computersprachen
2.0 Vienna, Austria
3.0 http://www.complang.tuwien.ac.at/projects/forth.html
ftp://ftp.complang.tuwien.ac.at/pub/projects/forth.html
4.0 university
4.1 Institut fr Computersprachen, TU Wien
5.0 Anton Ertl
5.1 anton@mips.complang.tuwien.ac.at
6.0 free
11.0 There's also some Forth material that is not referenced on the page, in particular:
ftp://ftp.complang.tuwien.ac.at/pub/forth/
http://www.complang.tuwien.ac.at/forth)

1.0 The Mops Page
1.1 Mar. 95
2.0 Philadelphia, PA, USA
3.0 http://www.netaxs.com/~jayfar/mops.html
4.1 private
5.0 Jay Farrell
5.1 jayfar@netaxs.com
6.0 free web/ftp
8.1 The Mops language by Michael Hore. The Mops system, manual, and Doug Hoffman's Selection Framework are directly available from my pub directory. Other files and resources are linked from other sites via the web page.
10.0 My ISP's Unix boxes, which I connect to using a Mac Quadra 605
11.0 Mops 2.6 is Michael Hore's public-domain development system for the Macintosh. With Forth and Smalltalk parentage, Mops has extensive OOP capabilities, including multiple inheritance and a class library supporting the Macintosh interface.

1.0 Ron's Mac and Apple II archive
1.1 June 95
2.0 Milwaukee, WI, USA
3.0 http://141.106.68.98/
ftp:/141.106.68.98/
4.0 private
4.1 Ron Kneusel
5.0 Ron Kneusel
5.1 rkneusel@post.its.mcw.edu
6.0 free
7.0 ftp and http
8.0 10
8.1 Forth programs I've written for the Mac and Apple II.
8.2 yes
10.0 httpd4Mac-123a and FTPd 2.4
11.0 Types of files: pretty-printer for LaTeX, Forth on a simulated Apple II in Forth, microcomputer simulator/assembler, fractal-drawing program, CGI applications in Forth for MacHTTP.
To be added soon: Web Forms handlers for MacHTTP/WebStar; updated and "improved" Forth for the Apple Iie; simple program to show the period-doubling route to chaos.
Mac files are BinHexed Compact Pro archives (transfer as text); Apple II files are ShrinkIt archives (.shk, binary).

1.0 taygeta.oc.nps.navy.mil
1.1 1990
2.0 Monterey, CA, USA
3.0 taygeta.oc.nps.navy.mil (131.120.60.20)
www:
http://taygeta.oc.nps.navy.mil/fig_home.html
4.0 non-profit
4.1 Skip Carter
5.1 skip@taygeta.oc.nps.navy.mil
11.0 One of the premiere Forth archives on the net; includes the Forth Scientific Library, CD-ROM project, GEnie archives.

1.0 University of Bremen
3.0 //ftp.uni-bremen.de/pub/languages/

programming/forth
http://ftp.uni-bremen.de/FTP/ftp.html
5.1 ftp-admin@ftp.uni-bremen.de
11.0 Features a full ../Taygeta-Mirror archive (information from c.l.f post by dku@zarniwoop.cp-labor.uni-bremen.de (Dirk Kutscher).

Internet Mailing Lists

1.0 FIRE-L
1.1 Sept. 94
2.0 global
3.0 subscribe:
listserv@artopro.mlnet.com
submissions:
fire-l@artopro.mlnet.com
5.0 Moderated by Rick Hohensee
5.1 rickh@cap.gwu.edu
11.0 The Fire-l Mailing List is for updates, discussions, debate, speculation, and announcements of Rick Hohensee's free-form FIRE specification.

1.0 MISC mailing list
3.0 Subscribe to:
misc-request@pisa.rockefeller.edu
Articles: misc@pisa.rockefeller.edu
5.0 Jeff Fox and Penio Penev
5.1 jfox@netcom.com (Jeff Fox)
Penev@venezia.rockefeller.edu (Penio Penev)
11.0 The MISC list is about all aspects of the new P21/P8/P32 and F21 Minimal Instruction Set technologies being developed by Charles Moore and his MISC associates.

1.0 The Win32For mailing list
1.1 Dec. 94
3.0 for list entries:
win32for@edmail.spc.uchicago.edu
for un/subscribe:
win32for-requests@edmail.spc.uchicago.edu
5.0 Carl Zmola
5.1 zmola@cicero.spc.uchicago.edu
11.0 Discussion of all Win32For issues, the Win NT/95 object-oriented Forth system from Andrew McKewan and Tom Zimmer.

Electronic Mailboxes

1.0 The Forth Online Resources Survey (FORL)
1.1 July 95
3.0 forl@artopro.mlnet.com
11.0 A permanent mailbox/index for tracking the ebb and flow of all publicly available Forth electronic resources.

1.0 Miller Microcomputer Services
1.1 Dec. 90
2.0 Natick, MA, USA
3.0 dmiller@im.lcs.mit.edu
4.0 business
4.1 Miller Microcomputer Services
5.0 A. Richard Miller
5.1 dmiller@im.lcs.mit.edu
6.0 free
7.0 Internet
8.0 none
9.0 none
11.0 We stock Forth-related books (some on sale) and MMSFORTH software. We support licensed users of MMSFORTH, FORTHCOM, FORTHWRITE, DATAHANDLER-PLUS for IBM-PC (MS-DOS and non-DOS/standalone). We provide PC-compatible consulting and hardware. Request our free e-mail brochure "MMSFORTH and Forth books."

Newsgroups, Conferences, et al.

1.0 comp.lang.forth
11.0 Usenet newsgroup, c.l.f is the premiere global Forth bulletin board. Articles from comp.lang.forth are archived at:
ftp://asterix.inescn.pt/pub/forth/news/

1.0 GEnie
11.0 GEnie is a BBS run by General Electric Information Services (GEIS). It has a Forth "RoundTable" with a bulletin board and library. For info, including local access numbers (not just U.S. and Canada), phone 800-638-9636. "As a user and worker on GEnie, I have found customer service to be very good."

World-Wide Web

1.0 FORTH, Inc. Home Page
1.1 June 95
2.0 Los Angeles, CA, USA
3.0 http://www.earthlink.net/~forth
4.0 business
4.1 FORTH, Inc.
5.0 E. Rather
5.1 erather@forth.com
6.0 free website
11.0 Site includes summary info and detailed data sheets for FORTH, Inc. products, Forth programming course outlines, application descriptions (some with photos), and links to other Forth sites. Material added periodically.

1.0 F-PC Homepage
1.1 May 95
2.0 Eugene, OR, USA
4.0 private
4.1 Fred Warren
5.0 Fred Warren
5.1 fwarren@gears.efn.org
6.0 Free dialup access for all Forth files
8.0 five Forth files
8.1 related to F-PC Forth for the IBM-PC
9.0 Mail
9.1.0 netmail
9.1.1 FTP
11.0 This home page is dedicated to the version of Forth for the IBM-PC known as F-PC. It is a full-featured, non-ANSI compliant, public-domain version of Forth—a superset of Forth-83 Standard. This page provides an introduction to Forth, an introduction to F-PC, downloading F-PC and tutorial material, and on-line mini-tutorials on using features of F-PC. This page will eventually be a repository for useful F-PC libraries.

1.0 Jeff Fox's Home Page
1.1 Dec. 93
2.0 Berkeley, CA, USA
3.0 http://www.dnai.com/~jfox
4.0 Business
4.1 Ultra Technology
5.0 Jeff Fox
5.1 jfox@netcom.com (most often)
jfox@dnai.com (supports Eudora)
8.0 40 files
8.1 Ultra Technology, Computer Cowboys, Offete Enterprises, MISC chips, P8, P21, F21, P32, parallel programming in Forth, and AI.
9.1.1 uuenc/decode (on the netcom account)
11.0 This web site is organized by subject from the home page listed above. Incl. individual home pages for my company, Ultra Technology (http://www.dnai.com:80/~jfox/ultra.html); Chuck Moore's company (cowboys.html); Dr. Ting's company (offete.html); and for Minimal Instruction Set Computers (misc.html); as well as for MISC chips like P8, P21, and my chip, the F21.
There are FORML Conference papers, and *FD* articles in html format. There is a copy of the first published article on Forth by Chuck Moore in 1970 (4th_1970.html). Many documents are available in html, .DOC, .ZIP, .PRN, .TXT, with some .EXE, etc. All files are cross-indexed in ultrafre.html, which is listed as "Free Files" on my home page.

1.0 Nick Francesco's Forth Page
1.1 Feb. 95
2.0 Rochester, NY, USA
3.0 http://raptor.rit.edu/Nick/forth.htm
4.0 Private
4.1 Nick Francesco
5.0 Nick Francesco
5.1 nick@rit.edu
6.0 free
7.0 Web Browser
8.0 5
8.1 Forth resources on the net
8.2 yes
9.0 none
11.0 The Sound Bytes Radio Show Home Page:
http://www.vivanet.com/soundbytes

1.0 Phil Koopman's Forth Mini-Page
1.1 July 95
2.0 East Hartford, CT, USA
3.0 http://danville.res.utc.com/Mechatronics/ads/koopman/forth/index.html
4.0 personal
5.0 Philip Koopman
5.1 koopman@utrc.utc.com
6.0 free
8.0 Personal Forth and stack machine

publications
11.0 In html as of July 1995:
- WISC CPU/16 patent cover page and block diagram.
- WISC CPU/32 (Harris RTX-4000) patent cover page and block diagram.
- Preliminary exploration of optimized stack code generation (*JFAR* paper).
- Brief introduction to Forth ("two-page" language overview).

1.0 Pocket Forth Home Page
1.1 June 95
2.0 Phoenix, AZ, USA
3.0 http://chemlab.pc.maricopa.edu/pocket.html
4.0 Private on a community-college-owned computer.
4.1 Chris Heilman/Phoenix College
5.0 Chris Heilman
5.1 heilman@pc.maricopa.edu
6.0 free/daytime access may be slow or limited
7.0 www only.
8.0 about 40
8.1 Pocket Forth
8.2 yes
9.0 Click a link to e-mail the author of Pocket Forth.
10.0 Mac OS
11.0 This site is maintained by the author of Pocket Forth and includes archives of software written in Pocket Forth, such as programming demos, applications, and unique CGI programs written in Pocket Forth.

1.0 Stephan J Bevan's Web page
3.0 http://panther.cs.man.ac.uk/~bevan/forth
5.1 bevan@cs.man.ac.uk (Stephan J. Bevan)
11.0 Up-to-date FAQ information on Forth implementations and books; e-mail maintainer to make suggestions, corrections, and additions.

1.0 The TUNES project
1.1 1995
2.0 Paris, France
3.0 http://www.eleves.ens.fr:8080/home/rideau/Tunes/
5.0 François-René "Faré" Rideau
5.1 rideau@ens.fr
6.0 free (GNU copyleft)
8.0 Only part of one file points to Forth www sites, but the Forth spirit has contaminated the whole project.
8.1 Review of actual Forth in .../Review/Languages.html#FORTH and of my own version of Forth in .../LLL/LLL.html.
8.2 yes
10.0 SunOS 4.1.3
11.0 This site is for my TUNES system project, only remotely related to Forth. The only thing about actual Forth is: http://www.eleves.ens.fr:8080/home/rideau/Tunes/Review/Languages.html#FORTH

ADVERTISERS INDEX

Forth Vendors

To make suggestions, corrections, or additions to this list, contact:

Lyle Greg Lisle, P. E.
L Squared Electronics
2160 Foxhunter Court
Winston-Salem, No. Carolina 27106-9621
910-924-0629
L.SQUARED@GEnie.geis.com

Offerings codes:
L = Literature, S = Software,
H = Hardware, C = Consulting,
T = Training

Forth standards supported:
FIG = fig-Forth
F79 = Forth-79
F83 = Forth-83
ANSI = ANS Forth

4th Wave Computers Ltd.
C ANSI
2314 Cavendish Drive
Burlington, Ontario L7P 3P3 Canada
905-335-6844
p.caven@ieee.org

A Working Hypothesis, Inc
C
P.O. Box 820506
Houston, Texas 77282 USA
713-293-9484
70410.1173@Compuserve.com

AM Research
LSHC ANSI
4600 Hidden Oaks Lane
Loomis, California 95650–9479 USA
800-949-8051
sofia@netcom.com

Ampro Computers Inc.
H
990 Almanor Ave.
Sunnyvale, California 94086 USA
408-522-4825
techsupport@ampro.com

Bernd Paysan
S ANSI BigForth
Stockmannstr. 14
81477 MuenchenFRG Germany
++49 89 798557
paysan@informatik.tu-muenchen.de

Blue Star Systems
S ANSI Forth/2
P.O. Box 4043
Hammond, Indiana 46324 USA
ka9dgx@interaccess.com

Delta Research
S F83 JForth
P.O. Box 151051
San Rafael, California 94915 USA
415-453-4320
phil@3do.edu

FORTH, Inc.
LSHCT ANSI polyFORTH
111 N. Sepulveda Blvd. Ste. 300
Manhattan Beach, California 90266 USA
800-553-6784
ERATHER@aol.com

Forth Interest Group
SL
P.O. Box 2164
Oakland, California 94621 USA
510-893-6784
JDHALL@netcom.com

Frank Sergeant
SC ANSI Pygmy
809 W. San Antonio St.
San Marcos, Texas 78666 USA
F.SERGEANT@GEnie.geis.com

Frog Peak Music
S F83 HMSL
P.O. Box A36
Hanover, New Hampshire 03755 USA
603-448-8837
phil@3do.edu

L Squared Electronics
SC Pygtools, Pygmy
2160 Foxhunter Ct.
Winston-Salem, North Carolina 27106 USA
910-924-0629
L.SQUARED@GEnie.geis.com

Laboratory Microsystems, Inc. (LMI)
S F83 UR/FORTH
P.O. Box 10430
Marina del Rey, California 90295 USA
310-306-7412
duncan@nic.cerf.net

MicroProcessor Engineering Ltd.
HCLS ANSI PowerForth, ProForth
133 Hill Lane
Southampton SO15 5AF England
+44 1703 631441
sales@mpeltd.demon.co.uk

Miller Microcomputer Services
LSHCT F79 MMSFORTH
61 Lake Shore Road
Natick, Massachusetts 01760–2099 USA
508-653-6136
dmiller@im.lcs.mit.edu

Mosaic Industries, Inc
SH F83
5437 Central Ave Ste 1
Newark, California 94560 USA
510-790-1255

Mountain View Press, Div. of Epsilon Lyra
LSHCT ANSI MVP-Forth
Star Rt. 2, Box 429
La Honda, California 94020–9726 USA
415-747-0760
ghaydon@forsythe.stanford.edu

Offete Enterprises, Inc.
CHLST F83 eForth, F83 &
1306 South B St.
San Mateo, California 94402 USA
415-574-8250
tingch@ccmail.aplbio.com

Redshift Limited
S
726 No. Locust Lane
Tacoma, Washington 98406 USA
206-564-3315
RedForth@AOL.com

Rob Chapman
S botKernel, Timbre
11120 – 178 St.
Edmonton, Alberta T5S 1P2 Canada
403-430-2605
rob@idacom.hp.com

Science Applications International Corp.
CSTH ANSI Until, LMI, Uniforth
301 Laboratory Road
Oak Ridge, Tennessee 37831 USA
615-482-9031
smithn@orvb.saic.com

Silicon Composers, Inc.
H
655 W. Evelyn Ave., #7
Mountain View, California 94041 USA
415-961-8778

T-Recursive Technology
C ANSI
221 King St. East, Suite 32
Hamilton, Ontario L8N 1B5 Canada
905-308-3698
BJ@GEnie.geis.com

TOS Systems Inc.
C LMI
P.O. Box 81-128
Wellesley, Massachusetts 02181 USA
617 431-2456
rstern@world.std.com

Triangle Digital Services, Ltd.
H ANSI TDS2020 &
223 Lea Bridge Road
London, U.K. E10 7NE
+44-181-539-0285
100065.75@COMPUSERVE.COM

Ultra Technology
LSCT ANSI P21Forth
2510 – 10th St.
Berkekey, California 94710 USA
510 -848-2149
jfox@netcom.com

Vesta Technology, Inc
SHC ANSI, Forth-83+
7100 W. 44th Ave Ste 101
Wheat Ridge, Colorado 80033 USA
303-422-8088

(Letters, continued from page 5.)

- And Forth programs run fast because data is manipulated and passed in a common area, the data stack. The programmer has checked and debugged the use of this common area, and no run-time checking is required. (Data stack checking and debugging is probably the hardest part of Forth programming.)
- Forth programs are smaller than others because there are no checking and defining routines necessary.
- And Forth programs are smaller because data is manipulated and passed in a common area. Work areas (heaps) and work area managers are not necessary. The Forth programmer is the work area manager.

To conclude, Forth is an all-adaptable programming language usable by skillful programmers who understand the Forth programming language, the hardware, and the data they are using, and are capable of properly controlling all three. Many other programming languages are available for other people, but adaptations of Forth will never be one of them. Obviously, Forth cannot be the right language for everyone.

Should you and the others of FIG return and limit your interests to promoting the advancement of the use of the true Forth philosophy, I would be interested in rejoining.

Fred F. Kloman
Laguna Niguel, California

P.S. It has been impossible for me to believe that the people who were credited with such great intelligence have manipulated the path of FIG without seeing the great contradiction between the Forth philosophy and what they were doing. Forth is a very logical language, and a contradiction is an elementary logical situation. If they didn't see the contradiction, perhaps they are not as intelligent as they have been credited.

P.P.S. It would seem futile to attempt to recover interest in the real philosophy of Forth by publishing in *Forth Dimensions*. Very few of the many, many real Forth programmers of the world read the publication. We have all left FIG! And this explains FIG's hard times!

[See editorial on page 4 for commentary...]

Stretching Forth

Extending CASE by Simplifying It

Wil Baden
Costa Mesa, California

"Less is More"

In Forth, the definitions of @ ("fetch") and ! ("store") are independent from each other, and the two words can be used independently, although their uses are often paired. This is a characteristic of Forth—words are defined separately, and each word has an individual behavior. Words are not used together because of their syntax, but for what they do by themselves to the stacks and other data structures.

The definitions of the required control-flow words—IF, ELSE, THEN, BEGIN, WHILE, REPEAT, UNTIL, DO, LOOP, +LOOP, LEAVE, UNLOOP—are like the definitions of all the other required words. Each definition stands alone, independent of all the others. This independence is obtained by defining their behavior relative to a mysterious "control-flow stack" whose form and location are left unspecified.

There is no mention in the required words of "control structure." This is a recognition of how control-flow words have always worked in Forth.

THEN is not preceded by IF (and maybe ELSE) because of syntax, but because IF (and maybe ELSE) did certain things to the control-flow stack that THEN can use. The same can be said about the other required control-flow words.

In the optional control-flow words, this essence of Forth was overlooked, and the concept of "control structure" was introduced.

In particular, in the Core Extension wordset certain optional control-flow words were defined using "the CASE ... OF ... ENDOF ... ENDCASE structure."

Figure One shows formulations of CASE, OF, ENDOF, and ENDCASE that are coherent with the definitions of the required control-flow words. There is no concept of "control structure."

These words can be used wherever the Standard words can be used. However, they can also be independently mixed and matched, depending on the values in the control-flow stack.

With these definitions, OF can be used without CASE, and CASE can be used without OF. ENDOF is a synonym for ELSE.

Sample Implementation

In any system in which the data stack serves as the control-flow stack, the following is one possible implementation.

```
VARIABLE (CASE-MARK)
( This variable name should be
( kept hidden. )

: CASE
  (CASE-MARK) @   DEPTH (CASE-MARK) !
; IMMEDIATE

: ENDCASE
      POSTPONE DROP
      BEGIN
             DEPTH   (CASE-MARK) @ <>
      WHILE
             POSTPONE THEN
      REPEAT
      (CASE-MARK) !
; IMMEDIATE

: OF
    POSTPONE OVER    POSTPONE =
    POSTPONE IF    POSTPONE DROP
; IMMEDIATE

: ENDOF POSTPONE ELSE ; IMMEDIATE
```

Depending on how your system is implemented, other and perhaps better definitions could be made.

Examples

```
( "Thirty days hath September ...." )

: THIS-YEAR
  TIME&DATE NIP NIP NIP NIP NIP ;

9       CONSTANT SEPTEMBER
4       CONSTANT APRIL
6       CONSTANT JUNE
11      CONSTANT NOVEMBER
2       CONSTANT FEBRUARY
```

Figure One. The Simplified Case Statement.

6.2.0873 **CASE** **CORE EXT**

Compilation: (C: -- *case-sys*)

Mark the control-flow stack with an element to be used as a sentinel.

Execution: (--)

Continue execution.

6.2.1342 **ENDCASE** **CORE EXT**

Compilation: (C: *case-sys orig-1 orig-2 ... orig-n* --)

Append the execution behavior given below to the current definition. Then keep resolving the control-flow stack with the function of THEN so long as *case-sys* is not on top of the control-flow stack. Discard *case-sys*.

An ambiguous condition exists if THEN fails when doing this.

Execution: (*x* --)

Discard the top stack element and continue execution.

6.2.1343 **ENDOF** **CORE EXT**

Compilation: (C: *orig-1* -- *orig-2*)

ENDOF is an alternative name for ELSE.

See ELSE.

6.2.1950 **OF** **CORE EXT**

Compilation: (C: -- *orig*)

Put the location of a new unresolved forward reference on the control-flow stack. Append the execution behavior given below to the current definition. The behavior is incomplete until the forward reference is resolved, *e.g.*, by THEN or ELSE.

Execution: (*x1 x2* -- | *x1*)

If the two values on the stack are not equal, discard the top value and continue execution at the location specified by the consumer of *orig*.

Otherwise, discard both values and continue execution in line.

Note: OF is equivalent to OVER = IF DROP.

```
: DAYS                         ( month - days )
      CASE  SEPTEMBER        OF   30
      ELSE  APRIL            OF   30
      ELSE  JUNE             OF   30
      ELSE  NOVEMBER         OF   30
      ELSE  FEBRUARY <>      IF   31
      ELSE  THIS-YEAR 4 MOD  IF   28
      ELSE                        29
      0 ENDCASE
;
```

```
( Complex Multiple-exit Example )

: ROLL-FOR-POINT                       ( n - )
     BEGIN                             ( point)
           THROW-DICE          ( point n)
           DUP .
           7 OF DROP    LOSE  EXIT
           OF           WIN   EXIT
     AGAIN
;

( Note: OVER = IF DROP can be replaced
( by OF and vice versa. )

: CRAPS                                 ( - )
     THROW-DICE                   ( point)
     DUP .
     CASE  2 OF  LOSE
     ELSE  3 OF  LOSE
     ELSE  7 OF  WIN
     ELSE  11 OF WIN
     ELSE  12 OF LOSE
     ELSE  ROLL-FOR-POINT
     0 ENDCASE                          ( )
;

( For completeness, definitions of
( 'THROW-DICE', 'WIN', and
( 'LOSE' are given in the appendix. )
```

```
( Conditional Compilation Using a
( String Case Statement )

: [ELSE]                               ( - )
     1 BEGIN                       ( level)
       BL WORD COUNT     ( level word .)
       CASE
             2DUP S" [IF]" COMPARE 0=
       IF
             2DROP 1+
       ELSE
             2DUP S" [ELSE]" COMPARE 0=
       IF
             2DROP 1- DUP IF 1+ THEN
       ELSE
             2DUP S" [THEN]" COMPARE 0=
       IF
             2DROP 1-
       ELSE
             2DROP
       0 ENDCASE                    ( level)
       ?DUP 0=
     UNTIL                               ( )
; IMMEDIATE

: [IF]                           ( flag - )
  0= IF POSTPONE [ELSE] THEN ; IMMEDIATE
```

(Continues on next page.)

```
: [THEN] ( — ) ; IMMEDIATE
```

```
( Signum — negative/zero/positive
( discrimination. )

: SIGNUM                    ( n — -1|0|1 )
    CASE  DUP 0< IF DROP   -1
    ELSE  DUP 0> IF DROP    1
    0 ENDCASE
;
```

```
( Change carriage return to linefeed. )

    13 OF 10 THEN
```

Discussion

ENDCASE presumes that there is a test value still on the stack. This means that if you use that value between the last ENDOF and ENDCASE, you must DUP it first, or use it and restore a dummy.

In almost all applications, you want to do something with it.

CASE and 0 ENDCASE give a solution to an inconvenience with Forth control logic. Suppose that, despite your good intentions, you have a definition with nested IFs and ELSEs which end with many THENs.

Put CASE before the first IF, and 0 ENDCASE in place of the many THENs. This form is clearer, and it's impossible to miscount the THENs.

An example of such is a "string case" structure—see the definition of [ELSE] above.

The following may be a convenient definition.

```
: ESAC
  POSTONE FALSE    POSTPONE ENDCASE
; IMMEDIATE
```

Appendix

```
( Use your favorite Random Number
( Generator. )
( This one has an environmental
( dependency on 32-bit arithmetic. )
( Default RNG from the C Standard.
( 'RAND' has reasonable properties, plus
( the advantage of being widely used. )
VARIABLE RANDSEED
32767 CONSTANT MAX-RAND
: RAND                         ( — random )
     RANDSEED @                  ( random)
     1103515245 *     12345 +
     DUP RANDSEED !
     16 RSHIFT     MAX-RAND AND
;
: SRAND  ( n — ) RANDSEED ! ;   1 SRAND
: CHOOSE    RAND *         15 RSHIFT ;

: THROW-DICE
  6 CHOOSE 1+   6 CHOOSE 1+   + ;
: WIN        ." You win. " ;
: LOSE       ." You lose. " ;
```

Wil Baden is a professional programmer with an interest in Forth.

(Fast Forthward, continued from page 43.)

entering BUG would make visible just the two names SEE and DEBUG.

OS Maturity

Without a doubt, vocabularies increase the convenience and richness of the Forth development environment. However, they do not address all the needs that can be identified, including needs better served by modern operating systems.

A modern operating system allows running distinct applications in dedicated memory spaces. It can even afford them a certain amount of protection from corruption. It also permits easy loading and unloading of applications to let the user configure their preferred mix of instantly available tools (such as word processor, spreadsheet, etc.).

Forth supports instant access to mini-applications by letting you configure the Forth that comes up with your choice of preloaded mini-applications, or tools. However, the procedure is circuitous and often varies from one Forth system to another—even among several systems with a common host OS.

Furthermore, incremental changes are not well supported, except to load more tools.

Where Forth falls down is in its support for the incremental removal of one mini-application. The ability of Forth to conveniently *forget* (unload) a tool depends upon how recently it was loaded, and whether you don't mind also unloading any tools that happen to have been loaded more recently than it.

Such a simple task as unloading a ready-to-use application deserves an equally simple interface. Unloading of tools should not require the unintentional loss of executable code.

(Forth's view of compilation as the sole way to adjust the memory image is too narrow and too antiquated a view, as developers of Forth overlay managers already know.)

Recognizing this problem—and recognizing that a host OS underlies many Forth systems, system implementors have the opportunity to exploit the host OS to load or unload tools such as an editor. For a Windows-based Forth system, this provides a more convenient interface and makes the operation of Forth's tools more consistent with other tools on the same platform.

Assessing Vocabularies

Forth's vocabularies are serving a number of roles, as I have shown. Probably these roles are too numerous, suggesting that vocabularies are overloaded and therefore can't possibly perform well across the board.

When functioning as a means for changing focus between tools, vocabularies are satisfactory.

Corresponding GUI provisions can help you manage several concurrently loaded applications in a windows environment. Those GUI provisions include windows, application menus, taskbar-displaying utilities, and user-customizable menus—such as options for short and full menus. In future Forth systems, this particular application of vocabularies may be curtailed by taking advantage of superior GUI provisions.

When functioning as a way to change the configuration of tools that are loaded, vocabularies are not of any assistance as currently implemented. Forth's equivalent tools are suboptimal: We have tools for discarding (forgetting) compiled code and tools for regenerating an executable.

When functioning as a means of organizing source code, vocabularies are inadequate. Creating separate namespaces helps us isolate groups of routines for purposes of referencing them more precisely after they are defined. But before its compilation, the source code for Forth words is not subjected to any rigorous treatment that segregates them according to their vocabulary affiliation.

In any case, it may not be the role of formal language provisions to achieve such an objective. Code-structuring conventions may be more appropriate as a means to help us organize source code.

Vocabularies play a role like modules in terms of helping isolate groups of routines (and data) from other groups of routines, at least in terms of their visibility. But before vocabularies can be viewed as an effective substitute for modules, they require more development.

Nevertheless, vocabularies may be able to be integrated with other layers of software. Well integrated, external layers of software could augment and articulate vocabularies in various ways. By adding the right amount of outside support of just the right kind, an upgrade may be possible that offers much greater versatility.

Along with that, we may be able to better address how we can make compiled memory images more manageable. Recompilation alone is not enough. Recompilation is often unavoidable when, due to the unloading (or forgetting) of compiled code through operations that are not as granular as could be desired, more code was forgotten than was desired. (The unloading process has become even more constrained by the ANSI standard.)

Conclusion

Forth is a programming environment that is wide open. No other development environment permits a similar level of access to and modification of the tools for developing applications. For this privilege, we are willing to tolerate a certain amount of inconvenience. However, the rest of the programming world will not look upon this so kindly.

Let's acknowledge that vocabularies are overworked. New facilities should be introduced to handle the roles they do not serve well, or that they serve only in a peripheral sense.

Related problems should be attended to, as well. For example, we should strive to reduce the need for source code recompilation and kernel regeneration to just those occasions when the source code has changed. Currently, the need for such procedures arises due to system administration (system cleanup) activity. Let's give ourselves more convenient provisions to offload or rearrange memory-resident tools as part of our administration of a system.

One of the directions we need to explore is a form of compilation that permits vocabulary (or module) groups of words to occupy contiguous memory spaces. The natural next step is to compile such groups of words into execution units that can be relocated.

Such measures don't conflict with vocabularies, so an extension of vocabularies is one implementation option...

These measures could greatly improve the ease with which application- or module-resident memory is managed. Furthermore, such measures are not in conflict with the features of vocabularies. Therefore, an extension of vocabularies is one possible implementation choice.

(If the job can be done best by a host OS, perhaps the Forth kernel should become the equivalent of a shared library. That way, each application can be given its own address and stack spaces that are loaded and offloaded by the OS.)

Product Watch

FORTH, Inc. now publishes a home page on the Internet's World-Wide Web (WWW). The company's site at URL http://www.earthlink.net/~forth contains both brief and detailed product descriptions, application notes, and links to other Forth-related sites. These web pages also contain "mail-to" links so that web surfers can readily request information about how the company's tools support development of embedded systems, industrial controls, DOS-based real-time applications and Windows programs.

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

Vocabularies Are Overworked

Mike Elola
San Jose, California

One role that vocabularies serve well is setting the scope of name searches. In order to establish such search states, vocabularies also organize Forth words into groups. Each Forth word will have only one vocabulary affiliation.

The fact that words may be grouped into vocabularies should not be taken as evidence that the source code for each vocabulary is centralized in one place. Despite vocabularies, Forth source code can be haphazardly organized.

It might be enlightening to structure Forth source code more rigidly, such as by attempting to fix the location of various program elements. Other problems stand in the way of achieving this through formal language provisions, however (see the last installment of Fast Forthward). Vocabularies are among those Forth formalisms that are hindered from serving as effectively as they could as organizers of source code.

If the hindrances that impact our ordering of code were removed, the words in a vocabulary might be better organized in a file. Such a file could have at its start some code that declares an overall vocabulary state that remains in effect for the entire file.

Well integrated, external layers of software could augment and articulate vocabularies in various ways.

Scoping the Command User Interface

Forth is a strange and wonderful aggregation of tools. To shepherd these tools around, vocabularies play a substantial role. I will call this role one of *focus* management.

I am borrowing the term focus from the domain of user interface objects. GUI interfaces are populated with user interface objects that handle input events. As users navigate to an object, such as a text field or button, that object is said to have the focus. User interface events, such as keypresses, are handled by the object that has the focus.

Forth has a nongraphical user interface. User interaction results from typing something using the keyboard. Typically, we type commands with names that have particular meanings to us.

A Forth development environment might have one or more tools with identically named routines, however. Vocabulary search states permit one tool to take the foreground temporarily, while others tools are simultaneously hidden or pushed into the background.

This is a job for vocabularies. Vocabularies provide a means for Forth users to manage the system's focus. Systems such as F83 place vocabulary manipulation commands in a `ROOT` vocabulary, where they are readily accessible. (The fact that it was named the `ROOT` vocabulary should not imply that it is always the last vocabulary searched, however. At least, that is what I presume to be the case. Perhaps `ROOT` was not the best choice of names.)

By entering `EDITOR` (or `ALSO EDITOR`), you permit the editor words to take precedence over same-name words associated with other development tools. By entering `DOS` (or `ALSO DOS`), you permit file-manipulation words to take precedence over same-name words associated with other development tools.

By entering `FORTH` (or `ONLY FORTH`), you permit the focus to be narrowed to exclude all but the most basic development tools.

GUI menus typically contain commands for which keyboard sequences exist. Therefore, GUIs and command interfaces can share a common style of interaction.

Of course, focus management in Forth fails to parallel GUI user interfaces in all respects. The shift of focus in Forth through vocabularies is not as clear or intuitive as switching to another tool-dedicated window.

Forth consists of an aggregation of many tools into a single development environment. When you reference a vocabulary after `ALSO`, the system's focus widens to include the new tool as well any other tools that previously had the focus.

Comparing this with GUI interaction styles, it's as if the menus of several development tools were combined into one large menu bar. In such a way, Forth helps manage access to several simultaneously loaded mini-applications, each of which is typically a discrete development tool.

Forth permits many disparate commands to all be available at once. These commands might correspond to editors, debuggers, profilers, and so forth. In case any of those commands are named identically within different tools, your prior specification of the focus through a tool-oriented vocabulary can assure you of obtaining the command meaning you really want.

When using a GUI, keyboard shortcuts cannot be overloaded. However, by switching to another task window, you gain access to a new namespace for keyboard shortcuts. Only one window is active at a time, so the keyboard shortcuts must be unique in one application only, not across several applications. (Essentially, the same visibility limits apply within a vocabulary.)

When you use the Forth CUI (Command User Interface), commands are directly available. In contrast, a GUI will probably force you to choose the correct menu to go to first in order to find the command—unless you memorize the command shortcut.

Development Tool Deployment

Vocabularies help provide assured access to one tool at a time. Because vocabularies often are not exclusively used in one-to-one correspondence with development tools, they must play other roles as well.

Take, for example, the `USER` vocabulary in F83. It contains the words `VARIABLE`, `DEFER`, `CREATE`, and `ALLOT`.

The `USER` vocabulary supports tool development, assuring that each mini-application can incorporate per-user data structures. Private storage areas are helpful in a multitasking system so that users do not overwrite each other's work spaces when they run shared applications.

The `USER` vocabulary does not play the same role as does the `EDITOR` vocabulary. Its namespace need not come to the foreground or fade to the background to overtake control from or yield control to other development tools.

In order to qualify as a mini-application (or tool), a group of commands must accept input, process data, and produce an output. The words in the `USER` vocabularies serve another purpose, that of declaring specialized data structures.

Despite the different purposes that can be identified for vocabularies, vocabularies work their magic by affecting namespace search states. A couple of examples are: When `EDITOR` is excluded from the focus, Forth's editing tools become invisible; likewise, when `USER` is excluded from the focus, resources for writing multiuser Forth programs become invisible.

Consider how your car has component parts and how your toolbox contains discrete tools.

To make a particular repair, you need to use the correct tool. This corresponds to alternating between the tool-oriented vocabularies during Forth development, such as between the `EDITOR` and `DOS`.

To make a particular repair, you also need to obtain the correct parts. A Ford parts dealer is not the place to obtain a Chrysler part. This corresponds to selecting the correct vocabulary to obtain a domain-specific routine.

We may be tempted to think of domain-specific vocabularies in the same way as modules. They serve a module-like role, in that they help isolate one group of routines from another.

In his article "Understanding F83 Vocabulary Usage," Byron Nilsen listed nine vocabularies and gave short descriptions of each (see page 21 of *FD* XVI/ 1). Based on his descriptions, `ROOT`, `EDITOR`, and `DOS` appear to be the tool-oriented vocabularies. Of the remaining six, five appear to be domain-specific vocabularies. They are `ASSEMBLER`, `FORTH`, `HIDDEN`, `SHADOW`, and `USER`.

The remaining vocabulary, `BUG`, is ultimately domain-specific because it regulates access to a particular type of programming resource. It generally contains code-inspection words. Yet facilities for code inspection are truly tools that accept inputs and generate outputs, so there is impetus to classify `BUG` as a tool-oriented vocabulary. While most of the words needed to support `DEBUG` and `SEE` reside here, the `DEBUG` and `SEE` words themselves remain in the `FORTH` vocabulary. What happened?

Perhaps the words `DEBUG` and `SEE` are so few in number that a conflict of them and user interface words from other tools was not foreseen as a likely area of conflict, so there is no real need for a tool-oriented vocabulary.

Considering how the remaining words in the `BUG` vocabulary fail to comprise an application or development tool—it may have been viewed as a poor organizational strategy to group `SEE` and `DEBUG` together with them.

I don't think the notion that the developers of F83 might have been interested in avoiding extra typing when using `SEE` and `DEBUG` was a concern, particularly when `ALSO` is available. More likely, they wanted to preserve a private namespace for words that support code inspection, allowing more freedom to name words as they chose.

This illustrates a potential problem with the varied roles of vocabularies—one vocabulary might be pulled in several directions to enclose development-tool-oriented routines as well as domain-oriented programming provisions. Without further subdivisions, vocabularies cannot collect and distinguish both types of content.

A true module system has private and public parts. Likewise, a class or object system has a similar distinction between public messages and private implementations.

A few essays back in time, I suggested that an `INTERFACE` subvocabulary could be appropriate in certain contexts. The `BUG` vocabulary seems to be one where additional internal partitioning was needed.

Words in the `INTERFACE` subvocabulary could be searched at times when the words outside it but inside the same overall vocabulary remain hidden (or private). Those other words need to be searched, however, when extending a particular vocabulary domain.

Since I suggested this, I have had a suspicion that a better alternative might be a `PRIVATE` subvocabulary. Entering `BUG PRIVATE` would make available all the definitions that help support `SEE` and `DEBUG`. Whereas,

(Continues on page 40.)

CALL FOR PAPERS
FORML CONFERENCE

The original technical conference for professional Forth programmers and users.

Seventeenth annual FORML Forth Modification Laboratory Conference
Following Thanksgiving November 24–26, 1995

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California USA

Theme: Forth as a Tool for Scientific Applications

Papers are invited that address relevant issues in the development and use of Forth in scientific applications, processing, and analysis. Additionally, papers describing successful Forth project case histories are of particular interest. Papers about other Forth topics are also welcome.

Mail abstract(s) of approximately 100 words by October 1, 1995 to FORML, PO Box 2154, Oakland, CA 94621. Completed papers are due November 1, 1995.

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

Skip Carter, Conference Chairman   Robert Reiling, Conference Director

---

Advance Registration Required • Call FIG Today 510-893-6784

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

Conference attendee in double room—$395 • Non-conference guest in same room—$280 • Children under 18 years old in same room—$180 • Infants under 2 years old in same room—free • Conference attendee in single room—$525

Forth Interest Group members and their guests are eligible for a ten percent discount on registration fees.

Registration and membership information available by calling, fax or writing to:

Forth Interest Group, PO Box 2154, Oakland, CA 94621, (510) 893-6784, fax (510) 535-1295

Conference sponsored by the Forth Modification Laboratory, an activity of the Forth Interest Group.