$10 Volume XVI, Number 2 July 1994 August

# FORTH DIMENSIONS

SILICON COMPOSERS INC

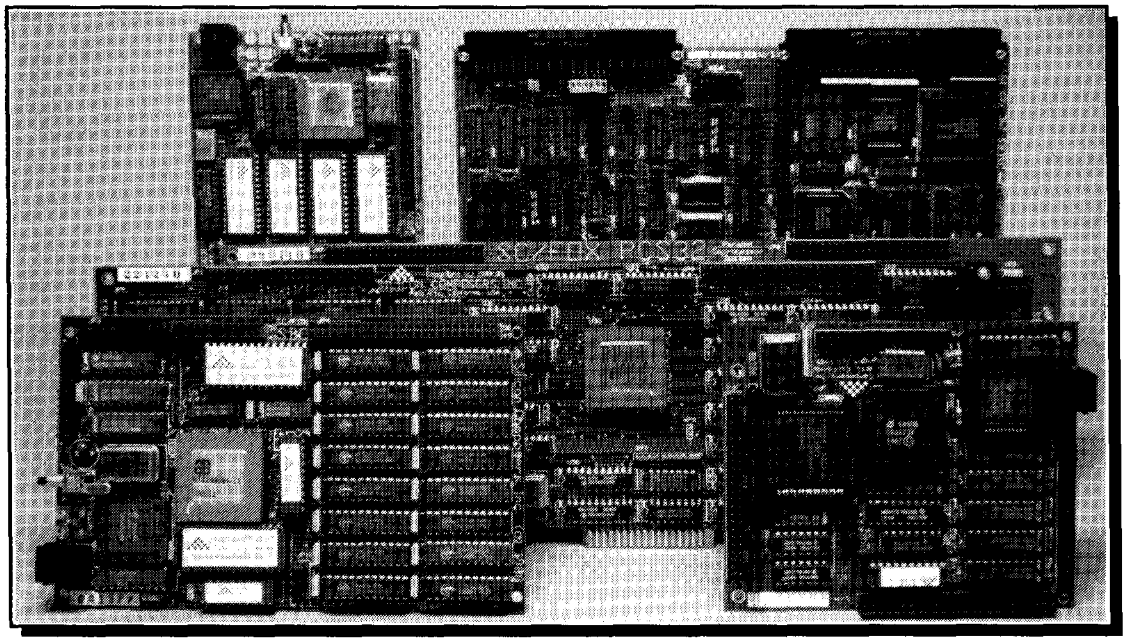
## *FAST* Forth Native-Language Embedded Computers

*DUP*

*>R*

*C@*

*R>*



**Harris RTX 2000tm 16-bit Forth Chip**
- •8 or 10 MHz operation and **15 MIPS** speed.
- •1-cycle 16 x 16 = 32-bit multiply.
- •1-cycle 14-prioritized interrupts.
- •two 256-word stack memories.
- •8-channel I/O bus & 3 timer/counters.

**SC/FOX PCS (Parallel Coprocessor System)**
- •RTX 2000 industrial PGA CPU; 8 & 10 MHz.
- •System speed options: 8 or 10 MHz.
- •32 KB to 1 MB 0-wait-state static RAM.
- •Full-length PC/XT/AT plug-in (6-layer) board.

**SC/FOX VME SBC (Single Board Computer)**
- •RTX 2000 industrial PGA CPU; 8, 10, 12 MHz.
- •Bus Master, System Controller, or Bus Slave.
- •Up to 640 KB 0-wait-state static RAM.
- •233mm x 160mm 6U size (6-layer) board.

**SC/FOX CUB (Single Board Computer)**
- •RTX 2000 PLCC or 2001A PLCC chip.
- •System speed options: 8, 10, or 12 MHz.
- •32 KB to 256 KB 0-wait-state SRAM.
- •100mm x 100mm size (4-layer) board.

**SC32tm 32-bit Forth Microprocessor**
- •8 or 10 MHz operation and **15 MIPS** speed.
- •1-clock cycle instruction execution.
- •Contiguous 16 GB data and 2 GB code space.
- •Stack depths limited only by available memory.
- •Bus request/bus grant lines with on-chip tristate.

**SC/FOX SBC32 (Single Board Computer32)**
- •32-bit SC32 industrial grade Forth PGA CPU.
- •System speed options: 8 or 10 MHz.
- •32 KB to 512 KB 0-wait-state static RAM.
- •100mm x 160mm Eurocard size (4-layer) board.

**SC/FOX PCS32 (Parallel Coprocessor Sys)**
- •32-bit SC32 industrial grade Forth PGA CPU.
- •System speed options: 8 or 10 MHz.
- •64 KB to 1 MB 0-wait-state static RAM.
- •Full-length PC/XT/AT plug-in (6-layer) board.

**SC/FOX SBC (Single Board Computer)**
- •RTX 2000 industrial grade PGA CPU.
- •System speed options: 8, 10, or 12 MHz.
- •32 KB to 512 KB 0-wait-state static RAM.
- •100mm x 160mm Eurocard size (4-layer) board.

For additional product information and OEM pricing, please contact us at:
**SILICON COMPOSERS INC 655 W. Evelyn Ave. #7, Mountain View, CA 94041 (415) 961-8778**

# Contents

# Editorial

**Forth Dimensions**
Volume XVI, Number 2
July 1994 August

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."

## Easier Done Than Said

Some find Forth easier to use than to describe. But we must speak intelligently and convincingly about Forth if it is to find understanding and acceptance among the uninitiated. The "Top 10 List" in this issue depicts Forth in terms familiar to many denizens of other, contemporary programming environments. It was prepared and distributed by the Forth Interest Group (FIG) at the 1993 Embedded Systems Conference. Since then, it has been reprinted in *Embedded Systems News*, AM Research's company newsletter. FIG chapters, Forth vendors, and others seeking permission to reprint it should contact FIG for details.

## X3J14: Done but Not Forgotten

Whether or not you are a Forth vendor or developer, be sure to read this issue's letter entitled "Forth's Three Problems," and the responses following it. You will find discussion of ANS Forth, the standardization process, and a proposal that FIG rather sweepingly endorse the new standard.

We invite you to further that discussion by submitting your own views and suggestions on the topic of ANS Forth and FIG's role in positioning it at the hub of the modern Forth community.

Mike Elola, *FD* columnist, FIG board member, and author of the "Top 10 List" and FIG's ANS Forth quick-reference guide, notes that the standard is a monumental piece of work which will be appreciated fully only if those who participated in its formulation will now step forward to point out its nuances, departures from past standards, and implications for Forth's future. How about it, X3J14 members? The best way to rally support around the fruit of your labors now is by helping us to understand it better. *FD* invites your contributions!

## Two-Way-Street Graffiti

We are pleased to present this 52-page issue of *Forth Dimensions*. It is the largest yet, thanks to authors and correspondents who, like you, are regular readers and FIG members. We encourage you to participate by writing articles and letters which help to shape the direction of this publication and its member-driven, parent organization.

We also wish to thank the individuals who generously support the Forth Interest Group in the form of contributions above and beyond the amount of the basic membership rate, and who introduce FIG and *Forth Dimensions* to friends and colleagues. Word-of-mouth recommendations, mention of Forth (including FIG's address!) in letters to other publications, and Forth vendors who include FIG literature in their customer mailings keep our organization alive and healthy, and will help to support expanded services and special projects benefitting everyone involved with Forth. Such generous, volunteer efforts are vital to our continuing success.

*—Marlin Ouverson*
*ouversonm@aol.com*

P.S. On the subject of generosity, check out the offer on page 11...

# Letters

*Letters to the Editor—and to your fellow readers—are always welcome. Respond to articles, describe your latest projects, ask for input, advise the Forth community, or simply share a recent insight. Code is also welcome, but is optional. Letters may be edited for clarity and length. We want to hear from you!*

## Mildly Eccentric

Dear Marlin,

Garth Wilson's claim in "Readability Revisited" (*FD* XV/6) that his Listing Two-b is "far more confusing" than Listing Two-a seems mildly eccentric, but to label Listing Two-b "The scrambled-eggs version" is a calumny. Far from being scrambled, it is rigorously systematic. Try modifying `SAMPLEWORD` by deleting one of the `THEN`s and changing the corresponding `ELSE` to `THEN` and it is immediately clear how the layout of Listing Two-b should be revised for the new definition. Conversely, a quick glance at the new layout will reveal how the pattern of control structures has altered. But with Listing Two-a it is far from obvious how the layout should be changed, if at all. The options seem limited to inserting or deleting blank lines. Listing Two-a distinguishes the various parts of the definition very tidily but says little about the relationships between them. Perhaps it is not so surprising if, in a very narrow sense, the layout of Listing Two-a seems less confusing than that of Listing Two-b, since it carries less information. But the missing information is still needed, and if it is not made apparent by the layout, we have to work all the harder to discern it at the semantic level.

There is one respect in which I prefer Listing Two-a to Listing Two-b. I have always felt that the Forth `IF` belongs on the same line as the conditions it tests. Now Garth Wilson has provided a principle to underpin this gut feeling—the zero-stack-effect line. I do not know if this is an original concept, but it is new to me and it is one of the best ideas I have read for a long time. It is very simple, yet it accounts for many examples of good Forth phrasing, which I previously assumed depended on subjective judgment. Like most good ideas, it seems glaringly obvious once somebody else has pointed it out!

Strangely, when Garth Wilson moves from the abstract example of Listing Two-a to the real code examples of Listing One and Listing Eight-b, he reverts to the common practice of putting `IF` at the start of a new line even though this contravenes the principle of the zero-stack-effect line. The effect of this on the program comments is significant. In both examples, a single Forth `IF` in the code requires two English `IF`s in the comments. If the zero-stack-effect line is used in the layout of `IF` structures, the wording of comments can be made much simpler and the code itself becomes correspondingly easier to comprehend, whether or not comments are actually provided.

The vertical alignment of related control structure words, the systematic indentation of control structures, and the zero-stack-effect line may appear to be mutually incompatible. But I remember reading several years ago of a layout strategy that can accommodate all three principles. It was probably in *Forth Dimensions* but I cannot give a reference. Briefly, it requires lines of program text to be right-justified (leaving a ragged left margin) and the end-of-line inside control structures to be offset to the left. This results in a kind of postfix indentation ideally suited to Forth. It looks a bit weird at first but, when you get used to it, it provides a very readable and very compact layout. Its main drawback is that, without specialized editing facilities, it can be tedious to write and maintain, which may be why it never caught on. If we stick with the more conventional, left-justified program text, something has to go. Garth Wilson is willing to abandon systematic indentation. I believe a better choice is to keep indentation, keep the zero-stack-effect line, and sacrifice vertical alignment.

Yours sincerely,
Philip Preston
London, United Kingdom

*Garth Wilson replies:*
Dear Editor,

I was very pleased that my article "Readability Revisited" in *FD* XV/6 has produced *three* letters so far. I don't remember ever seeing an article attract this number of letters in the time I've been a FIG member. Even if none of my recommendations had been accepted, the fact remains that the issue of readability is getting some desperately needed attention. Perhaps this "Letter to the Editor" section will serve as a forum for continuing the discussion even beyond this issue of *FD*.

While the subject of readability is in desperate need of attention, I was not terribly pleased with my treatment of it. My now infamous Listing Two seems to have struck out, since all three of the letter writers mention it negatively. Because of copyrights, I was not able to put in the examples I really wanted to use, so I only showed the form. Much of what I was getting at is given on page 85 of *Starting Forth* (2nd ed.), in the example of sorting eggs by size. This example would be a nightmare in the Listing Two-b format. (Perhaps the scrambled-eggs title I gave the listing was more appropriate than I realized.) Brodie ends five successive lines with `ELSE`, does not use indentation, and preserves vertical alignment. A few paragraphs later, he says, "Notice that the definition is visually organized to be read easily by human beings."

The factoring referred to by two correspondents is beautifully illustrated on pages 232–234 of *Thinking Forth* (original edition), in the example of the automated teller machine. One point I tried to make, however, is that factoring does not always work out so nicely. Sometimes the shortest truly descriptive name for the factor is almost a sentence, and anything shorter forces us to do the many levels of nesting mentally. We have to interrupt the definition we're reading to go figure out or review what the factor does. I've experienced this too many times. On page 183, one of Brodie's tips is to be sure you can name what you factor. On page 190, he says we shouldn't factor just for the sake of factoring. I don't think I'm being too bold to suggest that we not insist on factoring something that is not practical to factor.

When I follow the guidelines set out in the article, aging in those "cool dark places" (as Mr. Rottenkolber put it) has very little effect on my source code.

Tom Napier should be pleased that, in the article on interactive embedded software development, I was able to fit all the comments in without abbreviations and acronyms, and so use lower case. Since then, I found a programmers' text editor which will support my high-resolution monitor with up to 132 columns and 60 lines (a whole page!). It is Multi-Edit, by American Cybernetics in Tempe, Arizona. Now I shouldn't have to abbreviate anything. The difference is like getting a good seat at the ball game after having to watch it through a knothole in the fence. It also has a myriad of other nice features for programming.

Concerning his comment on using `THEN` instead of `ENDIF`, `THEN` seems to be almost as universal in Forth as `@`, `!`, `,`, and `#`, against which I have no complaints. If Mr. Napier likes `ENDIF`, however, I don't think any of us will complain.

Sincerely,
Garth Wilson
Whittier, California

### "Wordlists" in dp-ANS Forth

Dear Marlin,

In the preceding issue of *Forth Dimensions* (XVI/1), Mike Elola mentioned saving and restoring contexts in Forth by saving the values of variables on the stack. As an example, he mentioned that `LOAD` saves the state of `>IN` and `BLK` system variables on the stack, then restores them when done. This permits a block that is being `LOAD`ed to `LOAD` another block; after the second block has been `LOAD`ed and interpreted, interpretation of the first block resumes at the point where it was left off.

Mike went on to point out that a serious deficiency exists in terms of saving and restoring the vocabulary search order in the F83 standard, since there is no standard mechanism for accessing the current search order directly. (Of course, vendor implementations of F83 may include such a mechanism.) This could cause problems where words require the search order to be changed.

The new dp-ANS Forth, as described in Jack Woehr's book *Forth: The New Model*, includes a mechanism to save and restore the search order. `GET-ORDER` pushes a copy of the current search order onto the stack, in the form of a sequence of wordlist addresses followed by a count. `SET-ORDER` pops a similar count and sequence of wordlist addresses, and replaces the current search order with the sequence of wordlist addresses. ("Wordlists" are used in dp-ANS in lieu of vocabularies. A wordlist can be thought of as an unnamed vocabulary; `WORDLIST` creates a new wordlist, then returns its address on the top of the stack.)

Yours,
David M. Sanders
San Francisco, California

### Behind Bars

Dear Mr. Ouverson:

The definition of `ZCHKSUM` in "Print ZIP Barcodes" (*FD* XV/6) is not complete. If the least significant digit of the sum of the zip code's digits is zero, `ZCHKSUM` calculates a value of ten as input to `PZIP#`. It should calculate a value of zero.

A complete definition for `ZCHKSUM` would be:

```
: zchksum ( n -- ) 10 tuck mod - dup 9 >
    if drop 0 then pzip# ;
```

For example, BDS Software's zip code is 60025-0485:

6 + 0 + 0 + 2 + 5 + 0 + 4 + 8 + 5 = 30

30 10 mod —> 0 and 10 – 0 = 10

When 10 is added to `ZBARCODE`'s pfa in `PZIP#`, it points beyond the valid array and unexpected results occur.

USPS Publication 25 (page 24) gives the definition, "The correction character is always the number which, when added to the sum of other digits in the barcode, results in a total that is a multiple of 10."

Very truly yours,
M. David Johnson
BDS Software
P.O. Box 485
Glenview, Illinois 60025-0485

### Open Firmware and TILE, Forth at Both Ends of the Spectrum

Greetings:

I recently started working for a company which designs and builds processors and modules based on the SPARC architecture. Our product is used in notebooks, workstations, and super-computers. I was both surprised and pleased to find Forth alive and well here. Although most of the final testing and system software is written in C, Forth is still actively used during debugging, in the generation of special test cases, and in the development of the boot PROMs.

It turns out that Forth has a long association with SPARC-based computers. Mitch Bradley's Forth Monitor

*(Continued on page 47.)*

# Zero-Overhead Forth Interrupts

Garth Wilson
Whittier, California

In a previous article, I wrote about completely interactive embedded-systems software development on the target itself (*FD* XVI/1). I mentioned zero-overhead high-level Forth interrupt response, saying a description of that would have to wait.

A number of good articles have been published on providing Forth interrupt response. Without invalidating the work of others, I wanted to meet the challenge of accomplishing high-level Forth interrupt response in a much simpler way that would still work for most situations in typical indirect-threaded systems. The result is described here.

One man told me recently that he always does interrupts in assembly for speed, so he wasn't very interested in high-level interrupt service. The nice thing here is that when you eliminate the overhead, the speed increases substantially.

The zero-overhead interrupt support is very simple, adds only about 100 bytes to your overall code, and can be nested as many interrupt levels deep as you wish. No additional stacks are required. It's another natural for Forth. As usual for interrupts, some assembly is required, but very little.

I call it "zero-overhead" interrupt response because when an interrupt occurs, the Forth system moves right into the interrupt-service routine just as if it were part of the normal code. It is not necessary to first save any registers or prepare to use a different stack. Here's the summary: it is as if a new word was suddenly inserted into the executing code—a word whose stack effect is `( -- )`.

To illustrate, suppose we had an interrupt service routine (word) which, for the sake of simplicity, only consisted of

```
: ISR ( -- )  1 COUNTER +!  SYSRTI  ;
```

and Forth was executing the 2 in the line

```
PRINTER  2 SPACES  BOLD_ON
```

when an interrupt was requested. The effect would be the same as if there were no interrupts and the line had said

```
PRINTER  2  INC_COUNTER  SPACES  BOLD_ON
```

where `INC_COUNTER` had been defined as

```
: INC_COUNTER  ( -- )  1 COUNTER +!  ;
```

like `ISR` above. As you can see, the main program and the interrupt can both execute without interfering with each other, even though they use the same stacks and other resources. It is not necessary to save anything before executing the ISR or restore anything afterward. The only exception is that the `SYSRTI` above is just an ordinary `unnest` (or `EXIT`, `;S`, etc.) which also restores the ability to accept interrupts if appropriate. You may even decide to omit the `SYSRTI`. If you use the `SYSRTI`, the semicolon after it has no effect at run time.

Since servicing the interrupt does not require saving things, the interrupt service routine does not need any more stack space than other Forth words. Assuming we already had enough stack space to run Forth normally, we shouldn't have to worry about running out just because of the interrupts.

The only possible drawback with this method is that a primitive cannot be interrupted. Whatever is requesting service must wait until the current primitive is finished. This would only be a problem *if* you have primitives that take a long time to execute, ***and if*** those primitives are used at the times interrupt service is requested, ***and if*** the interrupt can't wait that long.

Otherwise, consider that it will typically take many primitives to service the interrupt, and it would be an insignificant delay to wait for one primitive in the main program to finish executing. It typically takes far less time to finish the currently executing primitive than to do all the register-saving and other setups required by other methods of high-level-language interrupt service.

The only return-from-interrupt overhead that is almost necessary with this method is that of re-enabling interrupts. If you don't need this done on a return from interrupt, the interrupt service routine can be a normal colon definition, ending with the standard `unnest` which is compiled by `;` (semicolon), and there will be absolutely zero overhead for return from interrupt, too.

Here comes the assembly. We have to make some small changes in `NEXT` that basically amount to polling, and these changes slow down the Forth execution by about

**Listing One-a.** Original version of NEXT (no interrupt support).

```
NEXT:  LDY  #1       ; Load Y for indirect indexing. Next, load accumulator
       LDA  (IP),Y   ; with hi byte of cell pointed to by instruction pointer.
       STA  W+1      ; Store it in hi byte of word pointer.

       DEY           ; Decrement Y. Some primitives expect Y to contain 0.
       LDA  (IP),Y   ; Load accum with lo byte of cell pointed to by instruction
       STA  W        ; pointer, & store than in lo byte of word pointer.

       CLC           ; Start addition with carry flag clear.
       LDA  IP       ; Load accumulator with instruction pointer lo byte,
       ADC  #2       ; add two to it,
       STA  IP       ; and store it back where you got it.

       BCC  next1    ; If the addition above didn't cause a carry, branch around
       INC  IP+1     ; the incrementing of the hi byte. Otherwise, increment.
next1: JMP  W-1      ; Jump to where it says "jump indirect W", so we get a
;----------------    ; doubly indirect jump.
```

one-thirtieth (in my system). If the interrupt requests come often enough, this method will run considerably faster than other methods, since you don't have to pay a big overhead penalty.

In the F83 system where I have implemented this (with an eight-bit CMOS 6502 processor), a couple of machine-language instructions added to NEXT load a byte from memory while simultaneously examining it to see whether it is zero or not. A branch is taken if appropriate. The choices are either to continue on as usual in NEXT, or to load the word pointer with the interrupt vector instead of with the contents of the address pointed to by the instruction pointer.

Some of the time taken by the extra pair of machine-language instructions is saved by the fact that we only allow two values for the byte which is fetched to see if interrupt service is necessary. These are values we would have to load into the processor's Y register anyway, even if we could somehow execute the right part of NEXT without testing.

If there is an interrupt to service, the new part of NEXT also turns off the bit in memory which records that there is interrupt service due. This takes less time than incrementing the instruction pointer, and loading the interrupt vector into the word pointer requires no indirect addressing. This means that the nest (or DOCOL, etc.) instruction in the interrupt handler actually gets executed *sooner* than the next instruction in the main code would have been executed had there been no interruption.

My original version of NEXT (before interrupt service implementation) was right out of the public-domain fig-Forth 6502 assembly source listing. The code in Listing One-a is what it looked like. (I have put all the assembly example listings here in a format used by "normal" assemblers, and commented them profusely especially for those few readers to whom 6502 assembly language is total Greek.)

After the modification, NEXT looks like the code in Listing One-b. Notice how much shorter the code is for responding to an interrupt than for continuing on with the next instruction in the main Forth code. This makes the relative interrupt response time very short. If we were to increment the instruction pointer when going to the interrupt-handling word, then the latter would be *replacing* the next Forth instruction in the main code instead of *delaying* it.

You will need a piece of machine code at the address pointed to by the machine-recognized interrupt vector location. If interrupts are enabled, this piece of code will be executed like any short machine-language interrupt service routine as soon as the hardware interrupt-request line goes true and the currently executing machine-language instruction finishes. This code only needs to put a byte in memory which can later be tested by NEXT, and disable the machine interrupt response so that the same code doesn't get executed over and over. Mine looks like Listing Two.

Next, you will need Forth words that enable and disable interrupting. These will probably have to be primitives, since most Forths won't have any words to access the µP status register. I called them IRQOK and NOIRQ. Another primitive, IRQOK?, returns my interrupt-disable flag.

A byte in RAM called irqok? (lower case) is used as a flag to record whether or not Forth interrupts are being allowed. irqok? is checked by SYSRTI, my Forth return-from-interrupt word. When a peripheral requests interrupt, setirq (in Listing Two) disables further interrupting but leaves irqok? alone.

You will usually leave interrupts disabled while the Forth interrupt service word is executing, and re-enable them when the interrupt service word finishes. SYSRTI is nothing more than unnest preceded by a few machine-language instructions to examine the content of irqok? and set or clear the processor's interrupt-disable bit accordingly. If you don't ever need to change the value of that bit immediately upon return, you can omit SYSRTI and the service word can be like any other colon defini-

**Listing One-b.** NEXT modified for interrupt support.

```
NEXT:   LDY  irqnot  ; Load Y with 0 if interrupt requested, otherwise 1.
        BEQ  runISR  ; Branch if interrupt requested, else continue here.
                     ; Y=1 now for indirect indexing. Load accumulator
        LDA  (IP),Y  ; with hi byte of cell pointed to by instruction pointer.
        STA  W+1     ; Store it in hi byte of word pointer.

        DEY          ; Decrement Y to 0. Some primitives will need Y to be 0.
        LDA  (IP),Y  ; Load accum with lo byte of cell pointed to by instruction
        STA  W       ; pointer, & store that in lo byte of word pointer.

        CLC          ; Start addition with carry flag clear.
        LDA  IP      ; Load accumulator with instruction pointer lo byte,
        ADC  #2      ; add two to it,
        STA  IP      ; and store it back where you got it.

        BCS  inc_hi  ; If the above addition caused a carry, branch to increment
        JMP  W-1     ; hi byte of instruction pointer. Else you're done. Done
                     ; with two JMP's because a branch not taken saves a cycle.
inc_hi: INC  IP+1    ; Increment hi byte of instruction pointer.
        JMP  W-1     ; You're done.
;-----------------
                        ; Pick up here if interrupt was requested.
runISR: INC  irqnot     ; Set irqnot =1, meaning no further Forth interrupt
                        ; service requested after this yet.
        LDA  FIRQVEC+1  ; Load the word pointer with the address pointed to
        STA  W+1        ; by FIRQVEC , a user variable.
        LDA  FIRQVEC    ; Load hi byte first, then lo byte. FIRQVEC is a RAM
        STA  W          ; address which holds the Forth interrupt request
                        ; vector CFA.
        JMP  W-1        ; Jump to where it says "jump indirect W", so we get a
                        ; doubly indirect jump.
;-----------------
```

**Listing Two.** This registers the interrupt request for NEXT.

```
irqrouting:             ; Machine-recognized interrupt vector points here.
      JMP   (MIRQVEC)   ; Jump to address pointed to by my machine-language
                        ; interrupt vector (MIRQVEC), which is initially setirq.

setirq:               ; Use to record IRQ for NEXT. Put this address in MIRQVEC.
      STZ  irqnot     ; Record that interrupt was req'ed by storing 0 in irqnot.
      STA  tempA      ; Temporarily save accumulator in tempA to put back later.
      PLA             ; Pull saved processor status byte off of µP stack,
      ORA  #04        ; set the bit corresponding to interrupt disable,
      PHA             ; and push the revised status byte back onto the stack.
      LDA  tempA      ; Restore the accumulator content.
      RTI             ; Return from interrupt. µP status gets restored modified.
;--------------
```

tion. (If you do use SYSRTI, remember that it should be followed by the semicolon to make the compiler happy.) My SYSRTI looks like the code in Listing Three.

To allow multiple-nested interrupts, an interrupt service word must re-enable interrupts (by invoking IRQOK). If you choose to do this, you might also want to push or otherwise save the content of irqok? and change it. This is so each return from interrupt leaves the interrupt-disable flag in the appropriate state. Obviously, if the flag is put back to the way it was just before the interrupt, it will always allow interrupts again. This is what SYSRTI will give you unless there was something in the interrupt service word that turned off irqok?. The purpose of irqok? is to tell SYSRTI whether or not to re-enable interrupts.

With indirect-threaded code, the average Forth primitive takes about 80 clocks to execute on the eight-bit CMOS 6502 with no wait states. Since, on the average, an interrupt will hit in the middle of an executing primitive, and since NEXT is quicker at starting interrupt service than it is at normal code, the average interrupt response time will be about 90 clocks, or 9 µS at 10 MHz. This includes the time taken by the short machine-language routine pointed to by the machine interrupt request vector, MIRQVEC. Many of the slower microprocessors cannot respond this quickly even in machine language; so to do it in Forth with an eight-bit µP is excellent. 10 MHz is the fastest bus speed currently available on the 6502 from Western Design Center in Mesa, Arizona. This makes for about 125,000 Forth primitives per second. They will be introducing faster ones in the near future. There are also 16-bit versions (the 65816 and its derivatives) and WDC is developing a 32-bit version.

A Forth interrupt service routine that only looks at an asynchronous communications interface adapter (ACIA) might look like this:

```
: SYSIRQ  POLL_ACIA  DROP  SYSRTI ;
```

Since here we only have one possible source of interrupts, we can DROP the flag telling whether or not it was the ACIA that requested service. If we had several possible interrupt sources, our SYSIRQ might look like the code in Listing Four-a. Listing Four-b is an alternative that uses a support word. ?EXIT is just my word to factor out occurrences of IF EXIT THEN. Any prioritized polling of interrupt sources can be put or called between SYSIRQ and SYSRTI above.

Table One gives a summary of the changes and additions used to accomplish zero-overhead high-level Forth interrupt response. A list of requirements is first, followed by a list of enhancements.

If you have a processor with several interrupt inputs, each associated vector would put the appropriate interrupt handler address in the FIRQVEC variable.

If you have hardware that prioritizes interrupts and gives the processor a byte to read to determine the source of an interrupt without polling, it may be appropriate to have a look-up table to convert the byte into a CFA of an interrupt handler.

Hopefully it won't take too much head-scratching or meditation for this to all make sense. It really is quite simple as high-level interrupts go; and if multiple nesting doesn't make it irresistible, the elimination of overhead and separate stacks certainly should.

**Listing Three.** Forth return-from-interrupt.

```
CODE SYSRTI             ; Lay header & code field down.
        SEI             ; Start with interrupting disabled.
        LDA  irqok?     ; Load & test byte at addr IRQOK? to see if IRQs are ok.
        BEQ  unnest+2   ; If not ok, don't execute next (CLI) instruction.
        CLI             ; Else clear interrupt disable flag.
        BRA  unnest+2   ; Branch to body of unnest (1st adr after code field).
;------------------
```

**Listing Four-a.** Interrupt-handler that polls potential interrupt sources.

```
: SYSIRQ
   POLL_TIMER     NOT  IF
   POLL_ACIA      NOT  IF
   POLL_KEYBOARD  NOT  IF
   POLL_PRINTER  DROP  THEN  THEN  THEN
   SYSRTI   ;
```

**Listing Four-b.** Alternative with a support word.

```
: POLL   POLL_TIMER      ?EXIT
         POLL_ACIA       ?EXIT
         POLL_KEYBOARD   ?EXIT
         POLL_PRINTER     DROP   ;

: SYSIRQ        POLL      SYSRTI  ;
```

Garth Wilson began programming in Fortran and assembly in college in 1982. Three types of BASIC and Forth were among the languages he later used for data acquisition and automated test equipment. He wrote the code for a flight-following computer in assembly. Much of his early programming was on a series of TI and HP hand-held programmables, which he used to facilitate a wide range of work. A friend told him a little about Forth in 1985, but it wasn't until 1989 that he picked up Brodie's book and started getting to know Forth. As a project to learn Forth, he wrote a cross-assembler and linker program. He enjoyed the language immensely, and was delighted to see development time plunge. Programming has been a part of his job since 1986. Now he is part owner of an aircraft communications company. He can be reached by phone at 310-695-7054 or by mail at 11123 Dicky Street, Whittier, California 90606.

**Table One.** Summary of new code.

| | |
|---|---|
| *Necessary:* | |
| `NEXT` | Inner interpreter. |
| `irqnot` | RAM byte to record whether or not interrupt pending. |
| `NOIRQ` | Primitive to set µP interrupt disable bit ( -- ). |
| `IRQOK` | Primitive to clear µP interrupt disable bit ( -- ). |
| `setirq` | Machine-language interrupt routine that puts 0 in `irqnot` so `NEXT` knows an interrupt was requested. |
| `SYSIRQ` | Secondary for actual high-level interrupt service. No special rules except that it usually will have `SYSRTI` just before the semicolon ( -- ). |
| `COLD` | (Modified, not new.) Before the first execution of `NEXT`, put 1 in `irqnot`, and make sure interrupts are disabled so you don't get into trouble before potential interrupt sources are set up. Invoke `IRQOK` and (optionally) set `irqok?` when Forth is ready to accept interrupts. |
| *Optional:* | |
| `IRQOK?` | Primitive to read µP interrupt disable bit ( -- f ). |
| `irqok?` | RAM byte to record whether or not to restore interrupt capability upon return from interrupt. |
| `SYSRTI` | Primitive (`unnest` version for return from interrupt) examines `irqok?`. |
| `MIRQVEC` | Variable containing an address used by the machine interrupt-service routine for a jump indirect. Not needed if you only have one routine. |
| `FIRQVEC` | Variable containing the Forth interrupt vector. If you have more than one high-level interrupt service word, put the CFA of one of them here. `NEXT` uses it to load the word pointer from in order to service the interrupt. |

**Special Offer for FIG Members Only:**

## SPECIAL DISCOUNT ON 16-bit polyFORTH!

For a limited time only, FORTH, Inc. is offering FIG members a chance to buy our 16-bit "segmented model" version of polyFORTH® for all DOS-based PCs at a drastically reduced price of only $295,* a *70% reduction* of our standard list price of $995!

Now is your chance to try this powerful system including data base tools, graphics, floating point, multi-user support, source for most functions, extensive libraries and utilities, and our outstanding documentation package.

This great, one-time offer made possible with the cooperation of the Forth Interest Group, includes:

- **Complete system**, including source (no Target Compiler®), all electives, and on-line Shadow Block documentation.
- **Documentation set**, including the *polyFORTH Reference Manual* (500 pp.), *80x86 CPU Supplement* (200 pp.), Programmer's Pocket Reference.
- **One month of free technical support** via FAX or Electronic Bulletin Board (1-day response)

**ACT NOW!** This offer will be good only until July 31, 1994. All you have to do is sign our Master Software License Agreement and send your check. **We also accept payment by MasterCard or VISA.**

Attractive discounts are also available on our other products, including our 32-bit "protected mode" polyFORTH for 80386/486 CPUs with GUI toolkit extensions; chipFORTH® for embedded microcontrollers; and our powerful, object-oriented EXPRESS™ system for industrial controls.

***CALL US TODAY* at 1-800-55-FORTH!**

***Ask for the FIG Special!***

* Plus $10 shipping (UPS Ground US & Canada) and applicable sales tax. For foreign sales, shipping billed at actual cost.

**FORTH, Inc.**
111 N. Sepulveda Blvd. #300
Manhattan Beach, CA 90266
800-55-FORTH 310-372-8493
FAX 310-318-7130

# Generation and Application of Random Numbers

Dr. Everett F. Carter, Jr.
Monterey, California

*Continued from last issue...*

## 6. Shuffling Numbers

Sometimes it is desired to randomize a small set of numbers so that a non-repeating sequence is obtained. An obvious application is in games, but there are other places where shuffling is useful. An example of this is in the oceanographic RAFOS float (Carter & Rossby, 1986). These freely drifting subsurface devices take measurements of the ocean for up to two years. After their measurement mission has ended, they go to the surface. They then broadcast their data to a satellite in 30-byte packets. Because of the limited lifetime of the instrument on the surface, and the fact that the satellites are not continuously above the horizon, it is possible that the instrument will fail on the surface before all the data is transmitted. Given the possibility that not all the data will be returned, it is best to have some data *throughout* the mission, and not just data from only one portion of the mission. To accommodate this, the packets are sent out in shuffled order, only repeating a given packet once all the packets have been sent.

When shuffling numbers, the important thing is to not repeat a number that has already been used. This means that taking the modulus of a generator such as `r250` won't work, because the numbers could repeat themselves.

A simple way to do this is described in Knuth (1981) as Algorithm P. The technique is to put the values to be shuffled into an array and to use a random number generator to generate indices into that array to actually shuffle the numbers. This array is then accessed sequentially to get the current number. A nice thing about this approach is that it will shuffle any predetermined sequence, not just a consecutive list of numbers. One just fills the array with the possible choices and runs `shuffle`. The example in Listing Three uses `ramp` to fill the array with consecutive numbers. The word `shuffle_test` in Listing Two demonstrates the algorithm.

## 7. Quasi-Random Numbers

The previous generators are all properly known as *pseudo-random* number generators—they all attempt to act like they are randomly picking numbers out of a hat. It turns out that for some applications pseudo-random numbers are a little *too* random. If you look back at Figure One, you will notice that there are places that are relatively undersampled and other places that have clusters of points.

If we change our generator so as to maintain a nearly uniform density of coverage of the domain, then we have a random number generator known as a *quasi-random* number generator. Figure Three shows a two-dimensional scatter plot of some quasi-random numbers. As you can see, the coverage has a distinctly different pattern from pseudo-random numbers. *Quasi-random* numbers give up serial independence of subsequently generated values in order to obtain as uniform as possible coverage of the domain. This avoids clusters and voids in the pattern of a finite set of selected points.

The generation of these maximally avoiding random numbers requires a good deal of bit twiddling. We will briefly describe here a rather efficient method using what is known as a Sobol' sequence. This method uses a set of binary fractions called direction numbers (these are the `iv` values in Listing Three).

The $j$th number is generated by doing a bitwise exclusive-or of all the direction numbers so that the $i$th bit of the number is non-zero. The effect is such that the bits toggle on and off at different rates. The $k$th bit switches once in $2^{k-1}$ steps so that the least significant bit switches the fastest, and the most significant bit switches the slowest.

The implementation shown of `quasi` in Listing Three (due to Antonov and Saleev, 1979), uses the *Gray code* of the number instead of the number itself. The use of the Gray code makes the generator very efficient. This is due to the fact that adjacent Gray codes differ from each other in just one bit position. This makes it possible to get the next quasi-random number by just doing *one* exclusive-or operation.

The direction numbers must be calculated according to the number of dimensions that the quasi-random numbers are to be used in. The word `quasi_init` is designed to calculate the proper direction numbers for up to seven dimensions. The mathematics behind the generation of the direction numbers is not trivial, but fortunately is not necessary in order to implement and use this generator. We will leave the mathematics to the references listed at the end of this article (Press & Teukolsky, 1989).

## 8. Monte Carlo Calculations

*The only good Monte Carlo is a dead Monte Carlo*
*—Trotter & Tukey, 1954*

The class of algorithms that solve problems probabilistically are known by the (purposely) colorful name of Monte Carlo methods. There are two types of Monte Carlo methods. One is the direct modeling of a random process (this is sometimes called *simple* Monte Carlo); queuing problems are a good example. The other class of Monte Carlo methods recasts deterministic problems in probabilistic terms (these are generally called *sophisticated* Monte Carlo methods). We will just focus on sophisticated Monte Carlo here.

While there are descriptions of something that could be described as a Monte Carlo method in the Bible, the method did not have any real practical application until computers became electronic. The first major application was solving neutron-diffusion problems during World War II (i.e., part of the calculations necessary to make the atomic bomb). The import of the statement of Trotter and Tukey is that Monte Carlo methods tend to be slower and less accurate than more traditional methods, *if there is a deterministic method to solve the problem.* When a deterministic method is developed to solve a problem, the Monte Carlo version generally turns out to be inferior.

Where Monte Carlo methods *are* used to advantage are: cutting-edge problems for which no deterministic method is known, problems involving a large number of dimensions (for which a deterministic method is either very time consuming or impractical to implement), problems involving complicated boundaries or other special conditions (which again may be difficult to implement using a deterministic algorithm), or problems where the needed solution is only part of the actual solution (say, finding only four or five out of 1000 unknowns). We will use small examples here for the purposes of illustration.

We will first look at using Monte Carlo to evaluate definite integrals. There are two major Monte Carlo techniques for evaluating such integrals. The first method is based upon an idea similar to the rejection method of generating random variables for arbitrary distribution functions. Suppose we wish to evaluate the integral,

$$I = \int_a^b g(x)dx \qquad (7)$$

If we put a bounding box around the function $g(x)$, then the integral of $g(x)$ can be understood to be the fraction of the bounding box that is also within $g(x)$. So if we choose a point at random uniformly within the bounding box, the probability that the point is within $g(x)$ is given by the fraction of the area that $g(x)$ occupies. The integration scheme is then to take a large number of random points with the box and count the number that are within $g(x)$ to get the area,

$$I \approx \frac{n^*}{n} V \qquad (8)$$

where, $n^*$ is the number of points within $g(x)$, $n$ is the total number of points generated, and $V$ is the volume of the bounding box.

This method is *very inefficient*, many points are required to make (8) converge towards (7) with any degree of precision.

A more efficient approach is to note that we can write (7) as,

$$I = \int_a^b g(x)f(x)dx = \frac{1}{V}\int_a^b g(x)f(x)Vdx \qquad (9)$$

if we define $f(x)$ as,

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is in the domain} \\ 0 & \text{otherwise} \end{cases} \qquad (10)$$

(again $V$ is the volume of the domain). (9) can be interpreted as the expectation of the function, $h(x) = g(x) f(x) V$, for the random variable $x$ which is uniformly distributed within the domain. This then gives an approximate procedure,

$$I \approx \frac{1}{n}\sum_i^n h(x_i) = \frac{V}{n}\sum_i^n g(x_i) \qquad (11)$$

Estimates based upon (11) converge much more quickly than those using equation (8).

If pseudo-random numbers are used for the Monte Carlo evaluation of integrals then, because of the clumps and voids in any given sample, there will be regions of the integral that are under-represented as well as over-represented. In the long run it is not a problem, since we know that the numbers represent a uniform distribution well. But "the long run" means using lots of iterations.

Probably the most effective way to speed up the convergence of Monte Carlo integration is to use *quasi-random* numbers instead of *pseudo-random* numbers for choosing the points. In general, this change will cause the integration estimate to converge towards the actual solution like $(\ln n)^N/n$ (where $N$ is the number of dimensions in the integral) instead of the usual $1/\sqrt{n}$. This improved convergence is considerably better, almost as fast as $1/n$.

The Forth word `mcint_test` in `RANTST.SEQ` demonstrates the use of (11) to evaluate the two-dimensional integral,

$$I = \int_0^1 \int_0^1 x^2 + y^2 \, dxdy \qquad (12)$$

A test of 5000 iterations (using quasi-random numbers) gives a value of 0.6664 (the exact value is 2/3). The same calculation using the same number of iterations with pseudo-random numbers (the code for this is not shown), gives an estimate of 0.6632.

Quasi-random numbers helped in improving integral estimates by attempting to cover the domain with a uniform density of points. Could we get even better results by sampling mostly where the function is large (where it contributes most heavily to the integral)? It turns out that the answer to this question is, yes. The general Monte Carlo technique of concentrating the sampling where the system is most influential is known as ***importance sampling***. By reducing the variance of the estimates, importance sampling can have a dramatic effect on the estimates. There are problems, however; for many problems we have no idea *a priori* where the regions that will contribute the most are at, and the problem of finding them can be as complicated as solving the original problem. When estimating an integral like (12) above, this is not a problem but we are faced with another difficulty: generating the sample distribution. Finding the inverse probability for the function is probably going to be at least as hard as solving the original problem, and using the rejection method to get the distribution is probably more expensive than just using quasi-random numbers in the first place.

One of the classic scientific applications of the Monte Carlo method is in the solution of differential equations. The cautions about the applicability of Monte Carlo, mentioned earlier, are especially important here. Using Monte Carlo to solve differential equations is very inefficient but, if there are special conditions, then it might be the best way to go. The idea here is to set up a random walk within the domain that the equation applies, starting at the point at which we want the solution. The probability of moving in each possible direction is determined by the differential equation that is being solved—it is not necessarily the same in each direction. The random walk continues until the particle reaches the boundary of the domain. At this point, the particle may be absorbed; the probability of this occurring depends upon the type of boundary condition that applies at that point. If the particle is not absorbed, the walk continues until it reaches a boundary and finally does get absorbed.

As a simple example, let us consider the steady-state temperature distribution of an annulus. Let's assume that the inner radius ($r = 1.0$) is held at a constant 40 degrees, and the outer radius ($r = 3.0$) is held at 60 degrees. The equation describing this situation is:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0 \qquad (13)$$

The standard finite difference approximation to this (assume the same size grid, $h$, in both $x$ and $y$) is,

$$\frac{(T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j})}{h^2} = 0 \qquad (14)$$

This can be rearranged to,

$$T_{i,j} = \frac{1}{4}T_{i+1,j} + \frac{1}{4}T_{i-1,j} + \frac{1}{4}T_{i,j+1} + \frac{1}{4}T_{i,j-1} \qquad (15)$$

Now we interpret equation (15) to mean that if at some time we are at position *(i,j)*, then at the next step we go to one of each of the four surrounding points with probability 1/4. This is the standard random walk.

So to solve the problem, we start out at the position we would like to have the solution at—say (2.0,0.0)—and do our random walk until we reach a boundary (because the mean squared distance from a starting point is linearly proportional to time, we will *eventually* reach a boundary).

For our example problem, the temperature is given on two boundaries, so when we reach a boundary we keep that value. We do many such walks, and our estimate will be the mean of the boundary values that we encountered. The program `laplace` implements this calculation (try it for a couple thousand walks), the value it returns can be compared against the exact answer $T = 40.0 + 20.0\, log(r) / log(3)$ where $r = \sqrt{x^2 + y^2}$ and X and Y are positions within the domain.

### Markov Chains

A Markov chain is a sequence of random values whose probabilities at a time interval depend upon the value of the number at the *previous* time. A simple example is the non-returning random walk, where the walkers are restricted to not go back to the location just previously visited.

The controlling factor in a Markov chain is the ***transitionprobability***, a conditional probability for the system to go to a particular new state, given the current state of the system. For many problems, such as simulated annealing, the Markov chain obtains the much-desired importance sampling. This means that we get fairly efficient estimates if we can determine the proper transition probabilities.

Markov chains can be used to solve a very useful class of problems in a way that seems almost magical. We will illustrate with the following problem: suppose we wanted to find the value of the vector $x$ that is the solution to,

$$x = Ax + f \qquad (16)$$

where the $n \times n$ matrix $A$ and the vector $f$ are known. By setting up a random walk through matrix $A$ we can solve for any single component of $x$.

A little mathematics is needed to see how this would work. First let's symbolically solve (16),

$$x = (I - A)^{-1} f \qquad (17)$$

This can be expanded to,

$$x = f + Af + A^2f + A^3f + \ldots = \sum_{m=0}^{\infty} A^m f \tag{18}$$

Now let's suppose we have an $n \times n$ matrix of probabilities, $P$, such that,

$$p_{ij} \geq 0 \qquad \sum_j p_{ij} \leq 1 \tag{19}$$

and we have an array,

$$g_i = 1 - \sum_j p_{ij} \tag{20}$$

further, we will define,

$$v_{ij} = \begin{cases} \frac{a_{ij}}{p_{ij}} & \text{if } p_{ij} \neq 0 \\ 0 & \text{if } p_{ij} = 0 \end{cases} \tag{21}$$

$P$ can then describe a Markov chain where the states of the chain are $n$ integers. The element $p_{ij}$ gives the *transition probability* for the random walk to go from state $i$ to state $j$. As long as $g$ is not zero, the walk will eventually terminate. The probability that the walk will terminate after state $i$ is given by $g_i$.

While taking the random walk, we need to accumulate the product,

$$V_m = v_{i_0 i_1} v_{i_1 i_2} \cdots v_{i_{m-1} i_m} \tag{22}$$

and the sum,

$$W_k = \frac{1}{g_k} \sum_m^k V_m f_m \tag{23}$$

The final $W$ value is important because its mean value (averaged over the walks that start at index i) is,

$$\begin{aligned} \bar{W} &= \sum_k \sum_{i_1} \cdots \sum_{i_k} p_{ii_1} \cdots p_{i_{k-1} i_k} v_{ii_1} \cdots v_{i_{k-1} i_k} f_{i_k} / g_{i_k} \\ &= \sum_k \sum_{i_1} \cdots \sum_{i_k} a_{ii_1} \cdots a_{i_{k-1} i_k} f_{i_k} \\ &= f_i + (Af)_i + (A^2f)_i + (A^3f)_i + \ldots \end{aligned} \tag{24}$$

Notice that the final form of (24) is exactly the $i$th element from equation (18). So to solve this problem we have three major steps:

- Set up the probabilities $p$ and $g$ and start off the system at the index at which we want to solve for $x$. Let's call that index $i$.
- Then we take a random walk until the walk terminates, accumulating the product $V$ and the sum $W$.
- Then we take the average of the $W$ values over several walks to obtain our estimate of $x_i$.

This will work as long as equation (18) converges; this will happen if the *norm* of $A$,

$$\|A\| \max_i \left( \sum_j | a_{ij} | \right) \tag{25}$$

is less than one (the smaller $\|A\|$ is, the faster the Monte Carlo estimate will converge). If the norm is larger than one, all is not lost, there is usually some manipulation that can be done to get a new matrix that has a small norm.

The code in Listing Four, `MARKOV.SEQ`, `row_solve` follows our above recipe almost directly. It first calls `init_p` to set up the probabilities. Then it calls `walk_chain` to take the individual random walks and accumulate the $V$ and $W$ values each time. Then it performs the averaging. Fortunately, it is easier to write the code than it is to wade through the mathematics that justifies the algorithm. These programs give reasonable results for 4000 or so iterations.

It turns out we can use this idea for all sorts of problems that have the same general form as (16). If write (16) as,

$$x = A(x) + f \tag{26}$$

and now consider $A$ to be *anything* that can operate on $x$ in a linear way, not just a matrix multiply. The mathematical jargon for such a beast is *linear operator*. Given the appropriate operator for a given problem, we can use the above method to solve several kinds of problems. We can do a matrix inverse, i.e., solve,

$$f = Hx \tag{27}$$

if we let $A = I - H$. Starting out at index $i$ will give us row $i$ of $H^{-1}$. The Forth word `row_invert` (Listing Four) does this calculation. If we restrict the chains to start at index $i$ and end at index $j$, then we obtain *a single element* of the inverse, $H^{-1}_{ij}$. This calculation is demonstrated in `element_invert`. Other problems that can be solved this way include the determination of eigenvalues and eigenvectors, and integral equations of the second kind such as,

$$x(t) = \int_a^b A(s,t)x(s)ds + f(t) \tag{28}$$

Notice that equation (28) has the same kind of form as equation (26), (integration is a linear operator). If we made a discrete grid upon which we wanted to solve (28) then we could use exactly the same code that we used to solve equation (16). However, in a practical application the dimension of equation (28) would be extremely large, or

*A(s,t)* would be so complicated to calculate that it is not really practical to create a giant matrix to approximate the integral. Instead we free up our random walk to apply *continuously* within the range [ *a,b* ]. Then the system is solved with a program that otherwise looks very much like `row_solve`.

## Conclusion

In this article we have looked at several uses of random numbers. Because of the different demands of applications upon the numbers, there is no one universal generator. The LCM generators are simple to code but they need care in choosing their parameters, are relatively slow, and their period is controlled by the size of the random numbers generated. Shift-register sequences like `r250` are fast and have large periods that are independent of the size of the random numbers, but `r250` has a large startup and storage overhead.

Quasi-random numbers prove to be useful for applications that need the distribution to be uniformly covered, as in Monte Carlo integration.

And, finally, we looked at applications where a randomly generated Markov chain can be used to solve problems involving linear operators.

## References

Antonov, I.A. and V.M. Saleev, 1979; USSR Comput. Math., Math. Phys., V. 19, p. 252.

Box, G.E.P, M.E. Muller 1958; "A note on the generation of random normal deviates," *Annals Math. Stat*, V. 29, pp. 610–611.

Binder, K. and D.W. Heerman, 1992; *Monte Carlo Simulation in Statistical Physics*, An Introduction, Springer-Verlag, Berlin, 129 pages.

Carter, E.F., and H.T. Rossby, 1986; "A Forth controlled oceanographic instrument," *J. of Forth Application and Research*, V. 4, No. 2, pp. 309–312.

Cooper, N.G., ed., 1989; *From Cardinals to Chaos*, Reflections on the Life and Legacy of Stanislaw Ulam, Cambridge Univ. Press, New York, 316 pages.

Knuth, D.E., 1981; *The Art of Computer Programming*, Vol. 2 Seminumerical Algorithms, Addison-Wesley, Reading, Mass., 688 pages.

McCracken, D.D, 1955; "The Monte Carlo Method," *Scientific American*, V. 192, No. 5 (May), pp. 90–96

MacDougall, M.H., 1987; *Simulating Computer Systems*, M.I.T. Press, Cambridge, Mass., 292 pages.

Maier, W.L, 1991; "A Fast Pseudo Random Number Generator," *Dr. Dobb's Journal*, May, pp. 152–157.

Park, S.K, and K.W. Miller, 1988; "Random Number Generators: Good Ones are Hard to Find," *Comm. of the ACM*, V. 31, No. 10, pp. 1192–1201.

Press, W.H., and S.A. Teukolsky, 1989; "Quasi- (that is, Sub-) Random Numbers," *Computers in Physics*, V. 3, No. 6, pp. 76–79.

Trotter, H.F. and J.W. Tukey, 1956; "Conditional Monte Carlo for Normal Samples," *Symposium on Monte Carlo Methods* (University of Florida, 1954), H.A. Meyer, ed., John Wiley, New York, pp. 64–79.

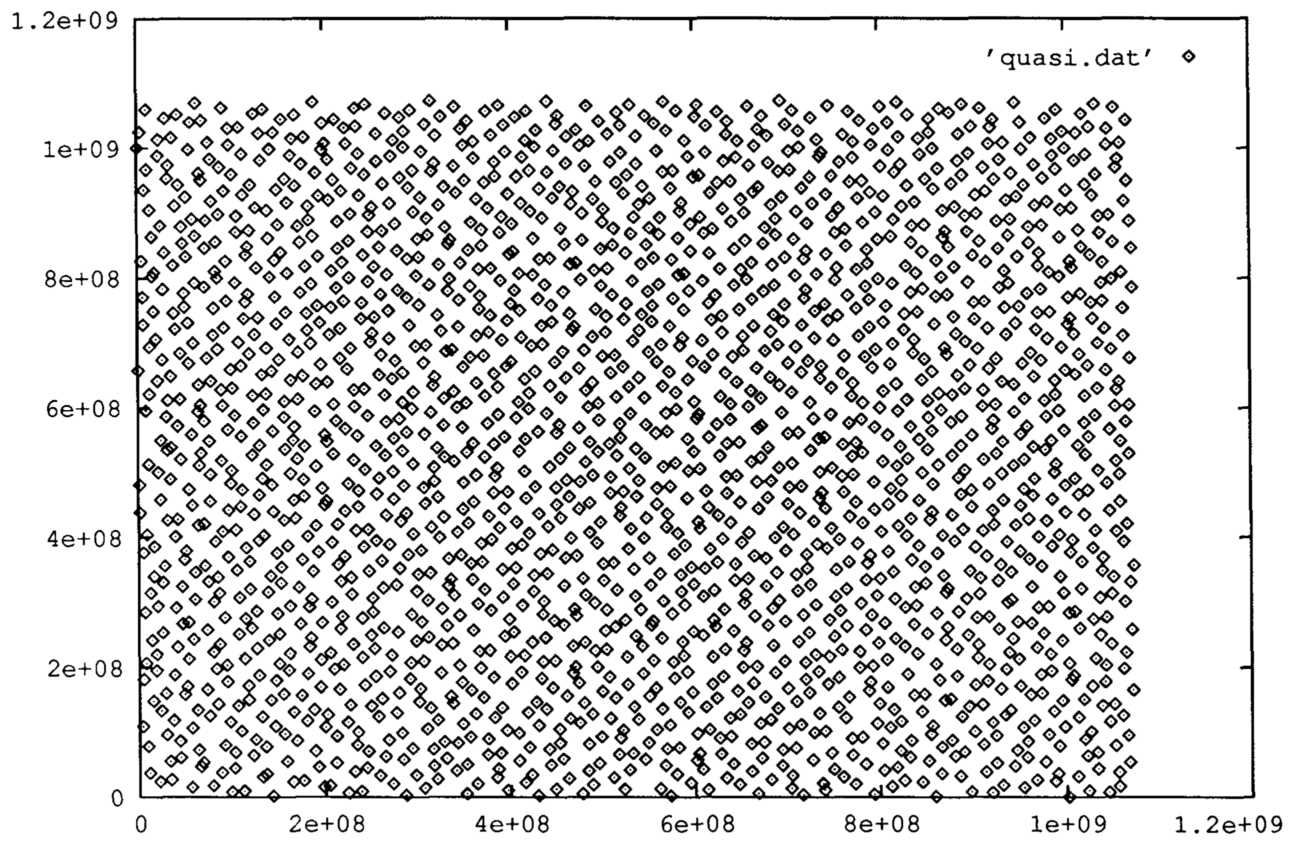
Rubinstein, R.Y., 1981; *Simulation and the Monte Carlo Method*, John Wiley & Sons, New York, 278 pages.

Everett Carter is an Assistant Professor of Oceanography at the Naval Postgraduate School. Prof. Carter wrote the Forth system for and helped design the RAFOS float which is being used internationally as part of the World Ocean Circulation Experiment. Back on land, he generates several billion random numbers a week running stochastic models of oceanic currents.

**Listing Three.** RANDS.SEQ

```
\ rands.seq     Integer Random number generation
\ NOTE:  32 bit doubles, uses DMULDIV.SEQ
\      (c) Copyright 1994  Everett F. Carter. Permission is granted by the
\      author to use this software for any application provided the copyright
\      notice is preserved.

: rand_task ;

needs dmuldiv.seq

cr .( RANDS.SEQ    V1.1                 1/18/94   EFC )

decimal

\ misc support stuff


: darray   create 4 * allot            ( n -- )
           does>  swap 4 * + ;         ( n -- addr )

: sarray   create 2* allot             ( n -- )
           does>  swap 2* + ;          ( n -- addr )
```

**Figure Three.** The two-dimensional scatter plot for 2000 points generated from a quasi-random number generator.



```
: dxor       ( d1 d2 -- d )             \ double xor
     rot xor -rot xor swap
;

: dor       ( d1 d2 -- d )              \ double or
     rot or -rot or swap
;

: dand     ( d1 d2 -- d )               \ double and
    rot and -rot and swap
;


65535.      2constant max16

2147483647. 2constant max32

2variable seed               1234. seed 2!

\ Linear Congruential Method, the "minimal standard generator"
\ Park & Miller, 1988, Comm of the ACM, 31(10), pp. 1192-1201
: lcm_rand   ( -- d )     seed 2@ 16807. umd*
                          2147483647. umd/mod
                          2drop
                          2dup seed 2!
;
```

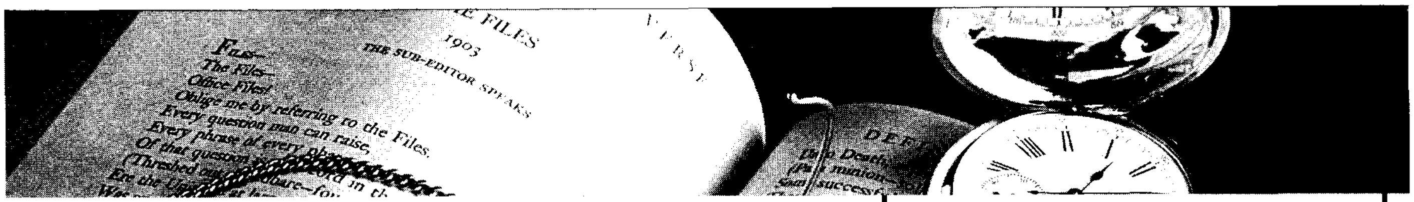
*(Continued.)*

```
\ Another linear congruential generator with poor choice of parameters
: lcm_bad ( -- d )         seed 2@ 1277. umd*
                           131072.   umd/mod
                           2drop
                           2dup seed 2!
;

: lcm_init ( d -- )
      seed 2!
;

defer lcm_32             ' lcm_rand is lcm_32

: lcm16  ( -- n )          \ 16 bit LCM random number
      lcm_32 drop
;

\ R250 code   --- 16 bit (unsigned) version
\ Kirkpatrick & Stoll, 1981; Jour. Computational Physics, 40, p. 517
variable r250_index
variable mask
variable msb
250 sarray r250_buffer

: r250_init   ( d -- )
     lcm_init  0 r250_index !

     250 0 do lcm16 i r250_buffer ! loop

     250 0 [ hex ]
           do
              lcm16 04000 >
              if i r250_buffer dup >r @ 08000 or r> ! then
           loop

    08000 msb !
    0ffff mask !

    [ decimal ]

    16 0 do
            i 11 * 3 + r250_buffer dup >r
            @ mask @ and
               msb @ or
            r> !
            mask dup @ 2/ swap !
            msb  dup @ 2/ swap !
    loop

;

: r250   ( -- u )

    r250_index @ dup 146 > if 147 - else 103 + then

    r250_buffer @
    r250_index @ r250_buffer @    xor

    dup    r250_index @ r250_buffer !

    1 r250_index +!
    r250_index @ 248 > if 0 r250_index ! then
;
```

```
\ R250 code -- 31 bit version
2variable dmask
2variable dmsb

250 darray r250d_buffer

: r250d_init ( d -- )
      lcm_init   0 r250_index !

      250 0 do lcm_32 i r250d_buffer 2! loop


      250 0 [ hex ]
          do
              lcm_32 020000000. d>
              if i r250d_buffer dup >r 2@ 040000000. dor r> 2! then
         loop

     040000000. dmsb 2!
     07fffffff. dmask 2!

    [ decimal ]

    31 0 do
         i 7 * 3 + r250d_buffer dup >r
         2@ dmask 2@ dand
            dmsb  2@ dor
        r> 2!

        dmask dup >r 2@ d2/ r> 2!
        dmsb  dup >r 2@ d2/ r> 2!
   loop
;

: r250d   ( -- d )               \ 32 bit positive (i.e. 31 bit) number

      r250_index @ dup 146 > if 147 - else 103 + then

      r250d_buffer 2@
      r250_index @ r250d_buffer 2@    dxor

      2dup    r250_index @ r250d_buffer 2!

      1 r250_index +!
      r250_index @ 248 > if 0 r250_index ! then

;

\ set the default 16 bit generator
defer rand_init            ' r250_init is rand_init
defer randgen              ' r250      is randgen
defer maxrand              ' max16     is maxrand

\ set the default 32 bit generator
defer rand_dinit            ' r250d_init is rand_dinit
defer drandgen              ' r250d      is drandgen
\                           ' max32      is maxrand



\ Quasi-random number generation (up to dimension 7)
\ Press & Teukolsky, 1989; Computers in Physics, V3, No. 6, pp. 76-79
```

```
 7 constant maxdim
30 constant maxbit
1073741823. 2constant quasi_max

variable dimension
2variable quasi_index
12 sarray ip
12 sarray mdeg
maxdim maxbit * darray iv
maxdim darray ix

: reduce_index ( i j -- k )
          1- maxdim * +
;

: bit_shift  ( n -- d )                    \ perform 1. << n
         >r 1. r> 0 ?do
                       2. d*
                  loop
;

: iv_normalize ( k j -- )
             maxbit over - -rot
             reduce_index iv dup >r >r
             bit_shift r> 2@ d* r> 2!

;

: Gray_Code  ( k j -- d )
        over over over
        mdeg @ -
        reduce_index iv dup >r 2@
        0. 5 pick mdeg @

        bit_shift umd/mod 2swap 2drop

        r> 2@ dxor

        3 pick ip @                  \ get ip[k]

        \ now do the "L loop"
        \ stack at this point:    k j xor_dbl ip[k]

        4 pick mdeg @ 1-          \ get mdeg[k] - 1
        dup 0 > if
          1 swap do
                  dup 1 and if
                            >r 3 pick 3 pick i - reduce_index iv 2@
                            dxor r>
                            then
                  2/
                -1 +loop
        else
                drop
        then

        drop rot drop rot drop

;

: quasi_init ( dim -- )             \ initialize for specified dimension

        dup maxdim > if ." quasi_init: dimension " .
```

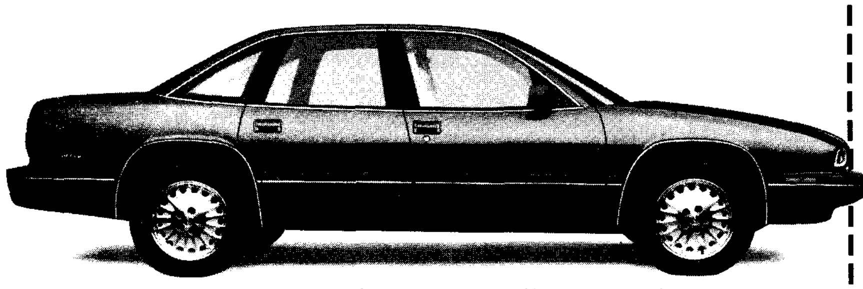
```
                          ." must be <= " maxdim . cr abort then

         dimension !

         maxdim 0 do 0. i ix 2! loop
         maxdim maxbit * 0 do 0. i iv 2! loop

         \ fill ip
         0 0 ip !   1 1 ip !   1 2 ip !   2 3 ip !   1  4 ip !   4 5 ip !
         2 6 ip !   4 7 ip !   7 8 ip !  11 9 ip !  13 10 ip !  14 11 ip !

         \ fill mdeg
         1 0 mdeg !  2 1 mdeg !   3 2 mdeg !   3 3 mdeg !   4 4 mdeg !
         4 5 mdeg !  5 6 mdeg !   5 7 mdeg !   5 8 mdeg !   5 9 mdeg !
         5 10 mdeg !  5 11 mdeg !

         maxdim 0 do .1 i iv 2! loop
         \ fill in the other elements of iv
         3.  7 iv 2!     1.  8 iv 2!   3.  9 iv 2!   3. 10 iv 2!
         1. 11 iv 2!     1. 12 iv 2!   3. 13 iv 2!
         5. 14 iv 2!     7. 15 iv 2!   7. 16 iv 2!   3. 17 iv 2!
         3. 18 iv 2!     5. 19 iv 2!   5. 20 iv 2!
        15. 21 iv 2!    11. 22 iv 2!   5. 23 iv 2!  15. 24 iv 2!
        13. 25 iv 2!     9. 26 iv 2!   7. 27 iv 2!
        17. 28 iv 2!    13. 29 iv 2!   7. 30 iv 2!   5. 31 iv 2!
        25. 32 iv 2!     3. 33 iv 2!  31. 34 iv 2!


          maxdim 0 do
                 \ normalize the set iv values
                 i mdeg @ 1+ 1 do
                        j i iv_normalize
                 loop


                 \ calculate the rest of the iv values
                 maxbit 1+ i mdeg @ 1+ do
                                \ calculate Gray code of iv
                                j i Gray_code
                                j i reduce_index iv 2!
                 loop

          loop


          0. quasi_index 2!
;

: quasi ( -- )                \ values returned in ix

         quasi_index 2@ 2dup 1. d+ quasi_index 2!

         maxbit 1- -rot
         maxbit 0 do              \ find rightmost zero bit
                 2dup
                 0= if
                     1 and 0= if rot drop i -rot leave then
                    else drop then
                 0. 2. umd/mod 2swap 2drop
         loop

         2drop
         maxdim *
        dimension @ 0 do
                dup i + iv 2@
                i ix 2@
                dxor
                i ix 2!
        loop

        drop
;

: .iv  ( -- )                        \ print iv values, useful for debugging
        cr
        maxdim 0 do
              maxbit 1+ 1 do
                          i 6 mod 0= if cr then
                          j i reduce_index iv 2@ ud.
                          loop cr
              cr
       loop
;

: .ix  ( -- )                        \ print ix values
      dimension @  0 do i ix 2@ ud. loop cr ;


\  the Shuffling algorithm, for producing a particular set of numbers
\  in random order, Knuth, 1981 (Vol. 2) Algorithm P

\  uses space at sh_ary to shuffle, change s! and s@ to use
\  some other location if necessary

52 sarray sh_ary                    \ shuffle up to 52 items
: s!    sh_ary  ! ;
: s@    sh_ary  @ ;

: choose  ( n -- m )                \ choose a value from 0..n-1 at random
        randgen swap mod
;

: exchange ( n1 n2 -- )             \ exchange element n1 and n2 in buffer
        over over
        s@ swap s@
        rot
        s! swap s!
;

: shuffle ( n -- )                  \ shuffles n elements
        dup 0 do
              dup choose i exchange
        loop
        drop
;

: ramp ( n -- )                     \ produces a simple increasing sequence
        0 do
          i i s!
        loop
;
```



## Study These Great Offers.

The smart money is on Alamo. Now you can enjoy $10 OFF any rental of three days or more or $20 OFF an upgrade on rentals of two days or more with Alamo's Association Program. And as always, you'll get *unlimited free mileage* on every rental in the U.S. In addition, you'll receive frequent flyer mileage credits with Alaska, Delta, Hawaiian, United and USAir. Alamo features a fine fleet of General Motors cars and all locations are company-owned and operated nationwide to ensure a uniform standard of quality.

As a member, you'll receive other valuable coupons throughout the year that will save you money on each rental. So study these offers and select the one that is best for you. For member reservations call your Professional Travel Agent or Alamo's Membership line at **1-800-354-2322**. Use **Rate Code BY** and **ID#** 378819 when making reservations.


*Alamo features fine General Motors cars like this Buick Regal.*

**$10 OFF A RENTAL OR $20 OFF AN UPGRADE**

- Certificate is valid for $10 OFF a rental or $20 OFF upgrade charges.
- Only one certificate per rental, not to be used in conjunction with any other certificates/offers.
- Certificate must be presented at the Alamo counter on arrival.
- This certificate is redeemable at all Alamo locations in the U.S.A. only. Once redeemed, this certificate is void.
- This certificate and the car rental pursuant to it are subject to Alamo's conditions at the time of rental. Minimum age for rental is 21. All renters must have a valid driver's license.
- This certificate is null and void if altered, revised or duplicated in any way. In the event of loss or expiration, certificate will not be replaced.
- Offer valid through July 21, 1994; except 02/17/94-02/19/94, 03/31/94-04/02/94, 05/26/94-05/28/94 and 06/30/94-07/02/94.
- No refund will be given on any unused portion of certificate. Certificate is not redeemable for cash.
- A 24-hour advance reservation is required. Reservations are subject to availability at time of booking. Valid on **Rate Code BY** and I.D. number 378819 only.

**If $10 Off a Rental is chosen:**
- Valid on intermediate through luxury car category.
- Offer valid on rentals of 3 days to 28 days.
- The maximum value of this certificate which may be applied toward upgrade charges is $10 off (not valid on time and mileage). No refund will be given on any unused portion of certificate. Certificate is not redeemable for cash.

**If $20 off an Upgrade is chosen:**
- Offer valid on rentals of a minimum of 2 days and a maximum of 28 days.
- The maximum value of this certificate which may be applied toward upgrade charges is $20 off (not valid on time and mileage). No refund will be given on any unused portion of certificate. Certificate is not redeemable for cash.
- Upgrade subject to availability at time of rental.

For reservations call your Professional Travel Agent or call Alamo at **1-800-354-2322**.

**U22B** Upgrade

**D61B** Rental

Alamo Rent A Car
Where all the miles are free

37876AS

**Listing Four.** MARKOV.SEQ

```
\ markov.seq    Examples of using Markov Chains
\     (c) Copyright 1994  Everett F. Carter. Permission is granted by the
\     author to use this software for any application provided the copyright
\     notice is preserved.

\       The true value of x for the problem x = A x + f is:
\                  7.5
\                  8.75
\
\       The actual matrix inverse for A is:
\                   2.8571    -1.4286
\                  -2.1429     3.5714

: markov_task ;

needs ffloat.seq
needs rands.seq

cr .( MARKOV.SEQ   V1.1                  1/25/94   EFC )

decimal

\ these next two are in STATS.SEQ and RANTST.SEQ and are repeated here

: fiarray   ( n -- )
        create dup , 0 do f0.0 f, loop
        does>
               swap dup 0<
                        if drop @
                        else
                           8 * 2+ +
                        then
;

: ranf ( f: -- num )
        drandgen float maxrand float f/
;

: fmatrix  ( n m -- )           \ defining word for a 2-d matrix
        create over , * 0 do f0.0 f, loop
        does>
               >r r@ @ * +
               8 * 2+ r> +
;

variable maxwalk
variable n
fvariable p

2 fiarray f
2 2 fmatrix a

\ rands_start is defered so that the actual initialization only
\ happens one time

defer rands_start

: rands_dummy  ;


: rands_init ( -- )

        ['] r250d is drandgen
        ['] r250d_init is rand_dinit
        ['] max32 is maxrand
        1234. rand_dinit

        \ only do rands_init once
        ['] rands_dummy is rands_start
;

' rands_init  is rands_start



: init_p ( n -- )
        dup dup * swap 1-
        ifloat ifloat f/ p f!

;

: adjust_a ( n -- )                   \ convert A to I - A
```

*(Listing Four continues on page 29.)*

# Top 10 List—Ways to Simplify Programming

## 10. Optimize the benefits of subroutines

**Choose a language that minimizes the run-time overhead of subroutines. That way, your use of many small, discrete, and reusable functions pays even greater rewards.**

*Forth is profoundly subroutine-oriented. Like most programming languages, Forth lets you create named subroutines and collect them into larger units known as programs. In Forth, programs are just subroutines set to run nonstop.*

*At Forth's center lies an engine for running subroutines. Forth systems are set up to run a command-interface routine at startup. Your commands cause other provisions to run, such as those that create (compile) and save new routines and those that locate and discard already compiled routines.*

Forth minimizes subroutine overhead, heightening the appeal of a modular programming style based on subroutines. Languages such as C set up and break down stack frames before and after each subroutine call. By avoiding this burdensome stack-management activity, Forth liberates subroutine calls from ungainly run-time overhead.

Like applications in other operating environments, Forth subroutines can be run as commands. This leads to quick turnaround times for each change-compile-test cycle. For a C subroutine to run as a discrete unit, source code must be altered and recompiled to meet the operating system's definition of an executable code resource. Through its incorporation of a run-time engine, Forth can dictate the forms executable resources may take. One of those forms is simple, efficient subroutines.

Forth can be characterized as a lean execution engine that is accompanied by an impressive collection of reusable subroutines. Each command you enter starts an execution sequence that can thread its way through many different routines. Execution threads direct the interplay of these routines so that various functions can spring to life, such as a compiler, an editor, and a host of other development tools.

## 9. Modularize down to the subroutine level

**Choose a language suited to a programming style involving many short subroutines that are easily understood, easily tested, and easily reused.**

*"[Forth's] modularity and other forms of error control allow production of remarkably bug-free application programs—perhaps more than any other language in common use."*
*—John James*

Forth lets you compose a long program as a moderate-length collection of short subroutines. For example, the broad functionality of Forth results from a collection of about one hundred subroutines, most of which call about seven others. When an execution thread is unwound, a lengthy execution sequence is often produced. Yet because each subroutine is defined and compiled only once, a compact, memory-conserving program can result.

A style of programming based on many small subroutines has been practiced by virtually all vendors of commercial Forth systems. Because of this, Forth provides access to a large number of useful and trivial-to-reuse data processing functions.

## 8. Scalability that does not sacrifice cohesiveness

**Choose a programming language that is seamlessly extensible and scalable, so the system is not disturbed by your extensions or cutbacks. Such a system remains cohesive despite its reconfiguration.**

Because Forth is subroutine-based, subroutines are the natural way to expand Forth or to scale it back. The Forth dictionary is designed to offer seamless expansion of the Forth environment. Of course, your newly compiled routines are accorded exactly the same treatment as the original, vendor-supplied routines. And by unlinking subroutines from the Forth dictionary, you can prune the dictionary and free memory.

*"Forth reduces the cost of a subroutine to very little, and the whole language is built on functions that are like subroutine calls. The programmer keeps defining new words (new functions) from old ones until, finally, one of them is the whole job." —John James*

Code resources are created in several ways. One way is to compile source code. Alternately, code resources may be supplied by the Forth vendor. However code resources came to be, a single Forth execution engine handles them exactly the same way. This regularity permits a run-time system that is small and efficient. Accordingly, applications derived from Forth can be pared down to very small sizes. Even large Forth systems can usually produce small applications to suit embedded applications (ranging between 2 and 30 kilobytes).

Forth is profoundly extensible. Unlike most languages, Forth extensibility is not limited to "functions," nor to any other restricted language element. For example, you can add new control-flow directives. For an embedded application, you might add a compiler directive that is equivalent to a C switch.

## 7. Syntactically lean programming languages

**Languages that are syntactically lean encourage you to create source code that is readily understood and easier to maintain. You should not have to think like a parser to untangle the meaning of source code! Forth shows how a language of homogeneous elements defeats the need for syntax-processing—both by humans and by computers.**

*"Never before in a high-level language has it been so easy to add new features, new data types, and new operators to a language. Unlike other languages, these new words (everything in Forth is called a word) have the same priority and receive the same treatment as words defined in the standard Forth vocabulary." —Greg Williams*

Forth's regularity eliminates the need for a parser to discriminate between different kinds of elements that may be deeply nested inside one another.

In contrast to the multiple-try algorithms of other compilers, the Forth compiler uses algorithms that are orders of magnitude simpler. A Forth system has the simple task of recognizing word-size elements only. (Forth routines named using one or more punctuation symbols are nevertheless the same homogeneous language elements as Forth routines named more descriptively.) A simple lookup table for subroutine names suffices for most of its needs.

## 6. Meaning directly conveyed by source code

**Choose a language that makes the meaning of source code obvious. Accordingly, programming systems should not steal any of your control over the evaluation sequence. To regain the control that should be exclusively yours, avoid languages with precedence rules and with evaluation-altering parentheses.**

You may already consider it a hindrance, this distasteful practice of peppering your source code with parentheses. This process is actually an encoding process, after which the meaning of source code is a step removed from your view. A substantial "decoding" step is required to account for precedence rules and for the locations of open and close parentheses.

*You frequently order operations to control their mutual effect. For example, you know better than to expect increment(print(10)) to print the value eleven. You understand how the parentheses elicit a right-to-left evaluation order. To obtain consistently left-to-right evaluation, the print function and all other functions must be made postfix, in which operands appear in advance of operations. The computer ultimately receives translated code ordered that way—as your debugger tells you. So why shouldn't you also see it that way? Why tolerate intrusive compiler actions that twist your code around?*

This affliction can be completely remedied through the liberal application of postfix notation. In Forth, you establish the order of evaluation by arranging the source code into the desired sequence. That way, the sequence you see is always the evaluation sequence you get (WYSIWYG). Because there is only one possible way it will be handled, Forth code is more likely to behave as you expect it to.

Computers require postfix ordering of operations. (An add operation is meaningless if the operands are not identified first.) When you adopt postfix notation too, your source code can more accurately convey your intentions on down to the computer chip doing the work.

Let's lay to rest the myth that you need computer assistance in this regard. You should be able to determine the meaning of source code without the hardship of nontrivial, code-resequencing rules. Parentheses are a nuisance rather than a convenience. This grievous type of activity leads to unexpected bugs, extra hours of debugging, and greater maintenance costs.

## 5. Seamless development and run-time environments

**Development environment tools should offer you a faithful preview of how code will behave when run in an unembellished target environment.**

The Forth user interface joins Forth's development and execution environments so seamlessly and transparently that a hybrid of the two is produced.

Nothing of substance distinguishes the execution environment for the target hardware and the execution environment used while you develop your application. The development system's execution environment is merely extended with more subroutines suited for development tasks. A core set of routines (a run-time system) travels with your applications. Those core routines are also present in the development system along with a fair number of additional tools providing services such as command handling.

Accordingly, test results obtained in the full Forth environment are much more trustworthy than those obtained using development tools that emulate the execution environment rather than incorporate it.

*"The whole system is written in Forth, right down to the bits—your application programs, the compiler, the operating system, the I/O drivers, etc. You do not have to learn some other language or be a systems specialist to modify it."*
*—John James*

## 4. Support for rapid development

**Choose a development system that lets you conveniently exercise recently compiled code, that supports incremental compilation, and that offers a way to minimize recompilation efforts. One consistent user interface for all development tools can also boost productivity by reducing errors and learning difficulty.**

To exercise recently compiled code without delay, you need a run-time facility that is continuously available as part of your development environment. The run-time engine must be able to respond to you, letting you determine when and how code you have compiled will run. Forth responds to commands you enter to compile or edit source code—and commands to run, debug, or discard compiled code. In these and other ways, Forth offers you a host of integrated tools for constructing programs rapidly.

With Forth tools, you can incrementally exercise and verify components of your application even when you have programmed only a small fraction of it, such as one subroutine.

Forth supports its own form of incremental compilation. First you compile and fix the lowest level of subroutines. Afterwards, you pay similar attention to the subroutines that rely on the ones you just refined. In the process, you discard only a portion of the previous compilation efforts—the portion you need to refine and recompile.

Forth comes pumped up with many tools for programming, including those for extending the compiler and those for compiling data structures. (The latter give you control over how values are actually stored in memory for variables and other types of data.) You can also extend the user interface or craft new tools to better facilitate development.

*"Forth also allows (and often encourages) programmers to completely understand the entire computer and run-time system. Forth supports extremely flexible and productive application development while making ultimate control of both the language and hardware easily attainable."*
*—Phil Koopman*

## 3. Friendly compiled-code tools (debuggers, etc.)

**Choose a development environment that provides powerful tools for inspecting and testing compiled code.**

When compiling a new Forth routine, the compiler stores the executable addresses of its constituent subroutines in memory (or an equivalent process, depending on the Forth you use). This makes the memory image of a compiled routine subject to meaningful inspection. By passing a decompiler an appropriate address to start examining, the Forth source code that led to the values stored in memory can be closely reconstructed. Likewise, the debugger's display of Forth execution flow can closely match source code listings.

## 2. Interoperability

**Choose a development system that takes advantage of your computer's operating system. That way you retain the help of popular tools and applications during development.**

Most Forth vendors offer systems designed to run under a host operating system. Many can take advantage of editors and code-management tools from numerous vendors. Recently, Forths have begun to appear that can use code resources (object files) created using other programming languages.

## 1. Forth

**Choose Forth. It is by far the most convenient way to meet the goals mentioned here.**

Most embedded systems programmers using Forth have little or no need for linkers, preprocessors, logic analyzers, in-circuit emulators, specialized debuggers, object-oriented tools and libraries, real-time operating systems, cross-compilers, and so on down the list of currently available tools.

Forth orbits closely around an efficient engine for subroutine execution. It has been adorned with just the right initial subroutines to provide functionality that you normally would associate with a much larger development system. Nevertheless, Forth's simplicity serves your needs while increasing your productivity.

Quotations from John James and Greg Williams originally appeared in the August 1980 issue of *BYTE* magazine.

*The non-profit Forth Interest Group helps support your use of Forth. FIG carries a complete line of Forth books, disks, conference proceedings, and other literature. Membership entitles you to discounts on books and conferences as well as a subscription to* Forth Dimensions.

To inquire about FIG membership benefits, call or write the Forth Interest Group at:
P.O. Box 2154 • Oakland, CA 94621 • 510-893-6784

(Listing Four, continued from page 24.)

```
        dup 0 do
            dup 0 do j i = if f1.0 else  f0.0  then
                     j i a dup f@ f- f!
            loop
        loop

        drop
;

: walk_chain ( n1 -- nlast, f: -- g )

        f1.0

        begin
          ranf
          p f@ f/   int drop dup
          n @ <
        while
          swap over a f@
          p f@ f/ f*
        repeat

        drop
;

: row_solve ( r -- , f: -- x )          \ solve x = A x + f  for element x[r]

          n @  init_p

          0
          f0.0

          begin
            over

            walk_chain

            f f@ f*
            f+
            1+ dup                  \ increment loop count

          maxwalk @ > until

          ifloat f/
          n @ ifloat f*

          drop
;

: element_invert ( r c --, f: -- inv )        \ solve for Inverse( A )[r][c]

          swap

          n @  dup  init_p  adjust_a


          0  f0.0
          begin
            over

            walk_chain

            swap 1+                     \ increment loop count

            >r >r over r> = r> swap     \ is last walk at desired column

            if f+ else fdrop then       \ if so, add to sum

            dup

          maxwalk @ > until

          ifloat f/
          n @ ifloat f*

          drop drop
;

: row_invert ( r -- )               \ solve for Inverse(A) row r
                                    \ result returned in array f

          n @  dup  init_p  adjust_a

          n @ 0 do  f0.0  i f  f!  loop

          0

          begin
            over

            walk_chain

            f dup f@ f+ f!

            1+                    \ increment loop count

            dup

          maxwalk @ > until

          n @ ifloat
          ifloat f/

          n @ 0 do fdup i f dup f@ f* f! loop

          drop fdrop
;


floats

: row_solve_init ( maxit -- )

        maxwalk !
        2 n !
        2 ifloat 0 f f!
        3 ifloat 1 f f!

        0.5 0 0  a f!     0.2 0 1  a f!
        0.3 1 0  a f!     0.4 1 1  a f!

;

: row_solve_test ( r maxit --, f: -- x )

        rands_start

        row_solve_init

        row_solve

;

: element_inv_test ( r c maxit --, f: -- x )

        rands_start

        row_solve_init

        element_invert

;

: row_inv_test ( r maxit -- )

        rands_start

        row_solve_init

        row_invert

;

: .f  ( -- )                       \ print array f

        cr 2 0 do i f f@ f. loop cr ;

: .a ( -- )                        \ print matrix a
           cr
           2 0 do
                2 0 do
                       j i  a f@ f. loop
                     cr loop
;
```

**Listing Five.** FILEIO.SEQ

```
\ fileio.seq              Words to make writing ASCII to a file
\                         simpler (uses the HANDLE code)
\     (c) Copyright 1994  Everett F. Carter. Permission is granted by the
\     author to use this software for any application provided the copyright
\     notice is preserved.

: fileio_task ;

needs ffloat.seq                    \ to handle input of floats

cr .( FILEIO.SEQ   V1.3                  1/4/94   EFC )

decimal

: ?wfail     ( t/f -- )
             0= abort" file write error" ;

: htype ( addr n -- )        \ for replacing type when writing to a file
        seqhandle+ hwrite
        ?wfail
;

: hcrlf     ( -- )
             2573 sp@ 2 seqhandle+ hwrite
             ?wfail
             drop
;

: its_true   ( n -- t )
        drop -1
;

: its_false ( n -- f )
        drop 0
;

: iswhite?   ( c -- t/f )

        case
            9 of -1 endof
           10 of -1 endof
           11 of -1 endof
           12 of -1 endof
           13 of -1 endof
           32 of -1 endof

           its_false
        endcase
;

\ skip to first non-whitespace, stores it at addr
\                                   n = -1 if file read error
\                                   n = count of whitespace skipped (0 if none)
: skipwhite ( addr -- addr n )

        0
        begin
          over dup
          1 seqhandle+ hread
          1 = if c@ iswhite?
              else drop drop -1 exit then
        while
          1+
        repeat

;

: hscan ( addr -- addr count )

         skipwhite

         0 < if
                0
         else
          1 begin
             over over + dup
             1 seqhandle+ hread
             1 = if c@ iswhite? 0=
                 else drop 0 then
           while
             1+
           repeat
           over over + bl swap c!       \ pad with a space at the end
        then
;

: hrewind ( -- )

        seqhandle+ hclose abort" file close error"
        seqhandle+ hopen  abort" file reopen error"

;


create iobuf    128 allot                    \ input data buffer

\ read a single float value from the currently opened file, put on fstack
: read_float ( -- n, f: -- x if ok )              \ 1 OK, 0 Fail
        iobuf 1+ hscan swap 1- c! iobuf c@ 0 >
        if
          iobuf fnumber
          floating? 0= if float then    \ if it parsed as a double (int) convert
          1                             \ it to a float
        else
          0                     \ zero indicates a failure
        then

;
```

# Forth Nano-Compilers

## for Microcontrollers and Beyond

*K.D. Veil and P.J. Walker*
*Sydney, Australia*

This paper describes a highly efficient microcontroller programming system based on Forth. The architecture of the particular target systems for which this was first developed demanded an unorthodox compilation method. During the process, it has become evident that this method could offer significant advantages in a wide variety of Forth systems.

### Introduction

The Forth programming paradigm exhibits a number of unusual and desirable features [1,2,3]. Forth is inherently simple, extensible, interactive, and produces very compact code, significant advantages in *any* environment but particularly for embedded microcontrollers. Unfortunately the architectures of many microcontrollers and other embedded devices (e.g., DSPs) do not lend themselves to interactive development; worse, the run-time overhead of Forth's address interpreter can impose an unacceptable penalty in time-critical applications.

***Most of Forth's shortcomings are a consequence of the implementation methods, rather than of the underlying language concept.***

Microcontrollers such as the 8051 family feature a strict Harvard architecture (code and data spaces are physically separate), and software development is typically based on cross-compilation and download of complete object code programs. This approach has limited interactivity but overcomes the run-time performance penalty inherent in native Forth implementations.

### Nano-Compilers—A Different Approach

We have developed an alternative approach to Forth object code generation, initially as a cross-compiler for the Intel 8051 family and subsequently for other processors, including the 8048, Z80, 80x86, and 680x0 families. Our approach makes each Forth keyword a small and highly specialised "nano-compiler" which generates the optimal target machine code to effect the function of that particular keyword. The current system has been implemented as a PC-based cross-compiler for a number of microcontroller targets and also includes a complete native code nano-compiler set for the host system.

If the target system is on-line via a serial port and running a minimal interface program, the incremental code produced can be automatically downloaded from the development host for immediate execution. It is thus possible, for example, to type `P1.0 off` at the host console with virtually immediate effect at the target port pin. Typically one would employ Forth extensibility to rename the relevant part according to its function and actually type `LED off` or something similar.

The target system is configured as a typical Forth two-stack machine [4,5], with the top items of both stacks held in dedicated processor registers for increased speed and efficiency [6]. All standard stack operators are defined and the target can be set to automatically return its stack contents to the host after each operation. It thus becomes possible to interactively program the target processor, to directly control peripheral devices, and to build definitions in the target memory. It is easy to forget and replace definitions, as is normal with Forth; it is also easy to compile standalone modules from source code and download to the target via the serial link. With these facilities, a complete application can be built word by word and debugged without the need for a hardware emulator. Object code for completed applications can be stored in host files (typically in Intel hex format) for production of firmware.

The example [given in Figure One] shows a typical mixture of immediate access to hardware devices and high-level definition. It also shows a number of features discussed in detail in later sections of this paper.

The change to nano-compilers is largely transparent to the user. If the code generated is not being compiled into a definition, it is compiled into a dedicated area for immediate execution after the entry of an end-of-line character. The traditional interactivity and extensibility of Forth have been retained but nano-compilation provides a number of significant advantages over threaded code interpretation.

As an example, consider the standard Forth word + used to add the top two stack items and leave the result on top of stack (TOS). If + is used inside a definition of a new word, traditional Forth systems will compile a reference (direct, indirect, or token) to the most recent definition of + found in the dictionary search path. In contrast, the nano-compiler + can optimise the code it produces on the basis of compile-time information. For example, if one or both of the numbers to be added at run time was entered as a literal, it is possible to improve run-time performance by compiling code appropriate to the situation.

The nano-compiler + for the 8051 target shows the basic principle. [See Figure Two.] Note that the top stack item is kept in the 8051's accumulator register to enhance execution speed.

The nano-compilers use an embedded assembler, written in Forth, to generate the target code. The example above uses the CPU-specific instruction set directly, and the system is therefore not portable. More advanced implementations [14] enhance portability by compiling to target machine code via a translation layer of generic operations pertaining to a two-stack virtual machine. This minimises the work required to port the system to a new target processor. It also allows each implementation to transparently take optimal advantage of the particular instruction set of its target CPU.

The compile-time optimisation principle can be extended to include automatic removal of redundant operations. As shown in the example above, top stack items are typically kept in CPU registers for increased execution speed; redundant register and memory shuffling can be easily detected at compile time and eliminated. The resulting code executes at speeds typical of "hand-tuned" assembler but with all definitions coded in high-level

**Figure One.** Typical mix of high-level and hardware access.

```
rename: P1.0 LED  ( for better readability)
3 constant: blink-frequency
12000 constant: crystal-frequency
variable: counter

crystal-frequency 12 / blink-frequency / constant: reload-value
( NB: all these calculations are performed at compile time! )

( Immediate instructions: )

LED on       ( user-friendly syntax )

P2.4 @
LED ! ( set LED to indicate
        port pin state )

counter @ last-count !  ( N.B. data-typed '@' )

counter @ LED !   ( note data type casting.
                  ( LED off only if counter 0 )

( In a definition: )

: counter-status ( - )
  " Total number of ticks is : " output
  counter @ output ;     ( high level word 'output'
                         ( is data typed according to
                         ( "class library" )
```

**Figure Two.** Nano-compiler for 8051 target.

```
: +   ( n1 n2 - n1+n2 )
  |DEPTH| case     ( special DEPTH; how many values
                   ( known at compile time? )
    0 of (SP) Acc ADD   ( None, compile TOS <- NOS+TOS )
        SP INC endof    ( remove NOS from stack )
    1 of           ( TOS is known )
      dup case     ( check its value )
        -1 of Acc DEC endof    ( -1 so use machine DEC )
         0 of         endof    ( it's 0 so nothing to compile )
         1 of Acc INC endof    ( +1 so use machine INC )
         Acc ADI  ( else some other value;
        compile immediate add )
      endcase
      +            ( else both known; execute + at
                   ( compile time )
  endcase ;
```

Forth. In fact, with the inclusion of interrupt-controlling words in the compiler described here, complete multi-interrupt embedded controller applications have been written without a single line of assembly code [8].

The automatic optimisation produces a simpler and more "programmer-friendly" interface. It relieves the programmer of the need to know special words like `1+`, `1-`, `2*`, `2/`, and also obviates "tricks" like `[ <number1> <number2> + ] LITERAL`.

The definition of nano-compilers in high-level code, such as the `case` conditional structure in the example above, enables the user to easily extend the optimisation if required. The user is also quite free to define new nano-compilers if the need arises, maintaining Forth's tradition of providing open and extensible systems.

Extensive user/programmer support is provided by the compile-time intelligence, for example to compare the stack comment of a definition with the stack effect in the definition code, or to check matching of `>R` and `R>` operations with warning messages to the programmer during compilation but with no run-time penalty. The repartitioning of compile/execute functions confers other advantages also, for example the removal of the normal constraint of conditional constructs (`IF` ... `ELSE` ... `THEN`, `BEGIN` ... `UNTIL`, etc.) only being able to be used inside definitions. If being used interactively, execution of compiled code is simply deferred until the conditional construct is complete. This feature also allows for conditional compiling without the need for special compiler directive words, providing a simpler and more intuitive Forth system.

A further powerful feature can be added to the system, using compile-time intelligence to effect "object-oriented data typing" [9]. Even in a relatively simple microcontroller application, several different data objects and their operations are likely to be used, for example bit, digit, byte, word, double, quad, float, as well as port and others (including special function register, or SFR, and external RAM on the 8051). Typically Forth implementations define special operators for different data types, so the data fetch operator will have a proliferation of variants: `B@`, `N@`, `C@`, `@`, `D@`, `Q@`, `F@`, `P@`, etc. Compile-time detection of object data type allows the compilation of appropriate code for the data type, but is invoked by the single orthogonal operator `@`. To achieve the performance required by real-time and embedded applications, it is essential that this processing occurs at compile time rather than run time [10, 11].

Rather than embodying the data-type decision process in a single nano-compiler `@`, different `@` nano-compilers can be defined in different vocabularies, one for each data type. This relieves the programmer from having to delve into a multiplicity of kernel definitions when defining a new data type. It also allows the use of a vocabulary search-order mechanism to provide easy inheritance of methods when defining new data types. By adding vocabularies of functions for, say floating-point or complex numbers, the equivalent of "class libraries" can be achieved.

The option is available to the programmer of casting the data type, for example by entering `W- @`, forcing the compilation of code for fetch from the word-type vocabulary (`W-`). The "operator overloading," normally associated with object-oriented programming, results in a significant simplification of the programming task without taking ultimate control from the programmer.

## Beyond...

This development was prompted, indeed forced, by the peculiar limitations of particular microcontroller architectures. Its usefulness has, however, extended well beyond those initial needs and we believe that the approach successfully addresses some of Forth's limitations, particularly in meeting the more general requirements of scientists and engineers [12].

A significant factor distinguishing scientists and engineers from, say, computer scientists is that whilst scientists and engineers are often familiar with the fundamental nature and operational principles of computers, they are generally more interested in computer programming as a means to an end rather than as an end in itself. In general, the programming environment should be as transparent as possible, providing a simple and direct linguistic bridge between problem and solution domains, but without being an inaccessible "black box."

The scope of scientific and engineering applications extends over a wide range—from, at one extreme, time-critical control of hardware interfaces to the manipulation, analysis and presentation of complex data structures at the other. It is more than possible that this full range would be needed in a single system, for example a custom data acquisition and analysis instrument incorporating an embedded microcontroller.

Forth perhaps comes closest to providing scientists and engineers with the ideal combination of access to underlying hardware, simplicity, and reconfigurability. It is very important in experimental and prototype systems to be able to develop and test methods quickly; many systems are constructed on a "one-off" basis with an inherently limited lifetime. Forth's inherent extensibility and syntactic freedom provide the possibility of specifying the solution to a problem in language close to the language of the problem itself rather than having to be translated via an intermediate (programming) language. Forth programs can produce highly self-documenting code, allowing high-level code sharing by non-expert colleagues. With Forth, one effectively builds the language to meet the problem rather than breaking down the problem into elements simple enough for a fixed language to deal with.

It has even been suggested that Forth could replace the scientists' outmoded mainstay, Fortran [13]. We do not intend in this article to attempt an analysis of cultural factors and fashion trends; factors which may partly explain Fortran's longevity as well as the recent ascendancy of other languages (particularly C) in engineering and science applications. We propose however that irrespective of these non-technical factors, Forth itself has shortcomings that limit its simplicity at the programmer/ user interface and impair its run-time performance in application to science and engineering problems. Datatyping complications, distracting stack manipulation, postfix awkwardness, run-time overhead, and poor facilities for linkage to non-Forth code collectively mean that Forth is still an unlikely competitor to more established programming environments. We do however consider most of these shortcomings to be more a consequence of the usual implementation methods rather than of the underlying language concept. We believe that the implementation of Forth we have outlined in this article may contribute to the ongoing development of Forth as a rich and powerful programming tool.

## References

1. Harris, K. "The FORTH Philosophy," *Dr. Dobb's Journal*, Sept 1981
2. Kogge, P. "An Architectural Trail to Threaded-Code Systems," *IEEE Computer*, March 1982, p22.
3. Brodie, L. *Thinking FORTH*, Englewood Cliffs (N.J.): Prentice-Hall, 1984
4. Malinowski, C. "A New 16-bit Realtime Controller sets new Performance Standards through Direct Execution of Forth," Harris Corp., 1988
5. *ANSI/IEEE X3.215-199x*: draft proposed ANSI standard document for Forth.
6. Veil, K. "THInC - An Interactive and Extensible FORTH Compiler," *Proceedings of the Australian Forth Symposium*, Sydney 1988
7. Hand, T. "Forth Readability," *Proceedings of FORML Conference 1984*, p83
8. Medical monitoring products CT100, CT200, CT300, CS2600, CS3000. Macquarie Medical Systems, Sydney Australia
9. Pountain, R. *Object-Oriented Forth*, Academic Press, London 1987
10. Paquet, E. "FORTH: Another Dimension," *Computer Language*, Vol 3, Dec 1986, p75
11. Noble, J.V. "Scientific Computation in FORTH," *Computers in Physics*, Sep/Oct 1989 p31.
12. Noble, J.V. *Scientific FORTH: A Modern Language for Scientific Computing*, Mechum Banks, 1992
13. Noble, J.V. "FORTRAN is Dead! Long Live Forth," *Journal of FORTH Application and Research*, Vol 5 No 2, 1988
14. Veil K.D. and Walker P.J. "Two-stack Virtual Machine Simplifies Forth Portability," *to be published.*

Klaus D. Veil has a particular interest in programming productivity and code efficiency. He has used Forth since 1981 in the design of products with embedded controllers, including a family of cardiac monitoring equipment, a range of ECG recorders, and a medical ambulatory monitoring system. Klaus is Senior Manager IT at Macquarie Health Corporation (Sydney, Australia) and past President of the Society of Medical and Biological Engineering.

Paul J. Walker has been a lecturer in physics in Australian universities since 1982. He has taught Forth in Physics Instrumentation courses since 1984, and uses Forth in solar energy and biophysics research projects. He was convenor of the Australian Forth Symposium held in Sydney in 1988.

*(Fast Forthward, from page 51.)*

While vocabularies give namespace management benefits, they stop short of offering private and public namespaces, and they stop short of managing routine interdependencies. Forth also lacks a standard way to perform module management operations such as those to specify module interdependencies, to load and release modules, and so forth.

Small Forth programs certainly don't need any more partitioning than is provided by a hierarchy of subroutines. But does such a means for reuse scale up well for larger programming tasks? Perhaps more ambitious Forth programs would be written if Forth had more provisions for partitioning and managing large programs.

Join me for similar ponderings (rethinking *Thinking Forth?*) in the next installment. The plot is bound to thicken due to the many subtle twists and turns inherent in Forth.

**FORTH and *Classic* Computer Support**

For that second view on FORTH applications, check out ***The Computer Journal***. If you run an obsolete computer (non-clone or PC/XT clone) and are interested in finding support, then look no further than *TCJ*. We have hardware and software projects, plus support for Kaypros, S100, CP/M, 6809's, PC/XT's, and embedded systems.

Eight bit systems have been our mainstay for TEN years and FORTH is spoken here. We provide printed listings and projects that can run on any system. We provide old fashioned support for older systems. All this for just $24 a year! Get a ***FREE*** sample issue by calling:

**(800) 424-8825**

**TCJ** The Computer Journal
**PO Box 535**
**Lincoln, CA 95648**

**ADVERTISERS INDEX**

| | |
|---|---|
| Alamo | 23 |
| The Computer Journal | 37 |
| FORTH, Inc. | 11 |
| Forth Interest Group | centerfold, 52 |
| LMI | 49 |
| MMS | 41 |
| Silicon Composers | 2 |

# Some Vulgar Functions

*Gordon Charlton*
*Hayes, Middlesex, England*

This article develops the ideas presented in the article "Rational Numbers, Vulgar Words" (*FD* XV/5) by presenting a small selection of higher functions and discussing their implementation.

A rational representation is of most use in areas of computation such as combinatorial analysis and probabilities which typically involve the use of simple fractions. Care still has to be taken to avoid extremely large or small intermediate results, but rounding errors, such as may be experienced with floating-point numbers, can be safely assumed not to occur.

Computations involving the use of irrational numbers are not any better, on the other hand, done with a rational representation than with floating point. Although in the best case a rational representation may represent an irrational quantity to higher precision than a similar sized floating-point representation, the worst case may be worse than floating point. As good floating-point routines exist for many of the higher functions, it is probably sensible to restrict the use of rational representations to those areas that they are unarguably best at.

**_A rational turtle, on the other hand, is quite appealing_**

It is not, therefore, very profitable to develop a full set of higher functions for a vulgar wordset. There are, however, a small set of operators which are of obvious use. In particular we note that there is much to be gained from the graphical representation of data. This need not be very sophisticated, and the familiar turtle graphics notation is quite suited to the task. I have found integer turtles to be unsatisfactory. (In part, this is to do with having had extensive experience with Logo prior to coming to Forth.) A rational turtle, on the other hand, is quite appealing. All that is needed for this is a vulgar sine and cosine function.

A review of work I have done in Logo and of those Logo books in my possession showed that, in addition to the arithmetic primitives, frequent use was made of square roots and random numbers. These, then, constitute (at least for my own personal use) a minimum acceptable subset of mathematical functions for a vulgar wordset. No doubt, with time additional functions will be added as and when required, but this is a good starting point.

## Vulgar Trig

Having identified not only a set of functions but also their primary usage, I am able to tailor them to suit that particular usage. In particular, graphical usage of trig very often requires both the sine and cosine of a given angle at the same time. Turtle graphic systems usually expect angles in degrees, rather than radians or some other representation which may be more convenient for the computer or the programmer. Finally, typical usage of turtle graphics is that the turtle is turned through simple fractions of a full circle.

Turtle graphics requires a very high degree of accuracy in its plotting routines, as users expect that, even after an extremely extended sequence of turns and moves, if the mathematics says that the turtle should be back at its starting position, then that is where it should be, not even one pixel out. Finally, the turtle should have a reasonable turn speed, which means fairly fast trig routines.

This is a fairly exacting specification, and meeting it meant a lot of code for something as simple as a sine function. (In fact, my one-screen development version grew six-fold meeting the spec.)

For speed, we need to drop out of a vulgar representation and into scaled integer; and for precision, we will use double-length numbers and rounding rather than truncation.

In the code that accompanies this article, I assume that the code given in the previous article is present on the system.

We start by looking at the two words `UFSIN` and `UFCOS`. These calculate sine and cosine, respectively, by Taylors series expansion. As they are set up initially, they are good for approximations between zero and one radians. We will use these to set up a lookup table for sines (and, hence, also cosines) between zero and ninety degrees in degrees, and further extend the range by making use of the symmetries of the trigonometric functions. UFSIN and UFCOS take their arguments as unsigned double-length integers, scaled so that the numbers they represent lie in the range 0≤d<1. (In other words, the high bit is 1/2, the next bit 1/4, and so on.)

The two preceding screens give us the functionality we require to use this representation. To multiply two such numbers, we multiply them as integers giving a quad-length result, and then rescale by discarding the low-order cells. (In fact, we do not discard them entirely, but use them to round off the result using "round to even" with Q>UF. Experimentation shows this does improve the accuracy of the results without significant loss of speed.) Next we need to be able to divide such a number by an integer, again with rounding. MU/R does this.

No special words are required for addition or subtraction, D+ and D- work just fine.

Finally, we need to convert to and from a vulgar representation. Converting to is done using the largest representable scaled number (all bits set) as an approximation to unity. (We need to shift these one bit to the right because REDUCE, the vulgar approximation word, expects signed numbers. This is done with rounding. Again, experimentation has shown this does improve results. You may notice that a rounded shifted *one* does still have the high bit set. I have taken advantage of my knowledge of the implementation of REDUCE here. I know I can get away with it!) Converting from is done without rounding, because experimentation showed it made no significant difference, and would have increased the amount of code required.

The words INT-SIN and INT-COS use the sine table to give very fast sines and cosines of degrees in the range 0 to 360 degrees. As the results are unsigned, the sign information is carried out by the back door, in the variables -SIN? and -COS?. This is a bit naughty, but convenient. These words are not intended to be reusable, they are merely steps on the way to the vulgar trig words.

Now we can screw down the effective range of UF-SIN and UF-COS by reducing the maximum number of times that they iterate. This is done by 3 SIN-LOOPS ! and 4 COS-LOOPS !. (These figures are for a 32-bit implementation, and will be smaller for a 16-bit system, as would the initial values of ten for both. To figure them out, add something like "I ." to UFSIN and UFCOS and count how many times they loop for an argument equal to one degree, in radians, before LEAVEing. The last trip round was unnecessary. You may find that you only need one or two trips round, in which case it is worth unraveling the loop, and defining words called, say, FRAC-SIN and FRAC-COS that just do the required. They are one-liners, and, as authors say when they can't be bothered to do the work themselves, are left as an exercise for the reader.)

Time for a bit of algebra:

```
sin (A + B) = sin A cos B + cos A sin B
cos (A + B) = cos A cos B - sin A sin B
```

These are the addition formulae for sine and cosine. By making use of them, we can compute both the sine and cosine of an angle almost as fast as we can compute the sine or cosine alone. This is our reason for dealing with the integral and fractional parts of the angle separately. Basically, the integral part is A and the fractional part is B. Having calculated sine and cosine of A and B once, we can combine them in two different ways to obtain sine and cosine of A+B. The word VSINCOS does this, along with converting from and to a vulgar representation, and between degrees and radians as required, and reducing the range to 0 to 360 degrees. For simplicity, I have mostly forsaken the stack in favour of variables. I'm not proud.

To finish off, I have given the basic trig functions in a more recognisable form for the sake of completeness, although, as noted above, it was VSINCOS I really wanted.

## Vulgar Square Roots

Although it is possible to derive square roots of vulgar fractions directly from their continued fractions, it is quicker to slip into scaled integers again, so this is what I do. First of all we note that;

```
sqrt (A / B) = sqrt (A * B) / B
```

This means we need only compute one square root. A little thought shows that the minimum necessary representation for best accuracy is scaled quad-length integer. This is achieved by the classic longhand method, for the simple reason that I have played with this a lot, so found it easy to adapt to deal with quad-length numbers. Note, however, that in order to provide a definition that runs equivalently on 16- and 32-bit systems, I have used a recursive definition that really hammers the stack. In 32 bits, in the worst case, it puts around 64 items on both the data and return stacks before bottoming out. It is an easy enough task to flatten out the recursion (an exercise for the reader!) at the cost of portability.

Note also that QSQRT cannot handle numbers with either of the two most significant bits set, because the remainder, which is accumulated as a double number, will overflow. This is not so dreadful. It means that, on a 32-bit system, the largest number it can handle is around 85 sextillion (85 undecillion in the American system) rather than 340 sextillion if all the bits were available. (For 16 bits, the numbers are 4.6 trillion—USA 4.6 quintillion—and 18 trillion.) If the third most significant bit is set, as it will be, the remainder can overflow on the way out of the routine, but this does not affect the result, other than destroying the possibility of rounding off. We do not need this facility to obtain best precision in the vulgar square root.

The word >TOP does the opposite of normalising a vulgar number, by ensuring it occupies as many bits as possible, short of the sign bit. This is done so that QSQRT will give as many significant bits as possible in its result. This form of scaling takes advantage of a different way of interpreting a vulgar number. Instead of thinking of it as a numerator and denominator, we can regard it as a scaled integer accompanied by its scaling factor, so 1/2 is one scaled by a factor of two, which is equal to hex 2000 scaled

## "Vulgar" Glossary

`: T* ( ud u--ut)`
Multiply unsigned double by unsigned single, giving unsigned triple result.

`: D= ( d1 d2--f)`
Return *true* if double d1 is equal to double d2, otherwise *false*.

`: Q* ( ud1 ud2--uq)`
Multiply unsigned double by unsigned double, giving unsigned quad result.

`: Q>UF ( uq--uf)`
Convert scaled quad to scaled double with rounding. Scaling is such that represented numbers are in the range $0 \leq x < 1$. Rounding is "round to even."

`: UF* ( uf uf--uf)`
Multiply unsigned scaled double by unsigned scaled double, giving unsigned scaled double result. See `Q>UF` for details.

`: MU/R ( ud u--ud)`
Divide unsigned double by unsigned single with rounding, giving unsigned double result. Rounding is to even.

`: T/ ( ut u--ud)`
Divide unsigned triple by unsigned single, giving unsigned double result.

`: M*/ ( ud u1 u2--ud)`
Multiply unsigned double by unsigned single u1, and divide by unsigned single u2, giving unsigned double result. Triple-length intermediate product is used to avoid overflow.

`constant ONE`
Nearest approximation to unity allowed in scaled system described in `Q>UF`.

`: UF>V ( uf--v)`
Convert scaled double (as above) to vulgar.

`: V>UF ( v--uf)`
Convert vulgar to scaled double (as above). Vulgar should have no integral component, and be positive.

`2constant PI`
Vulgar approximation to pi.

`: DEG>RAD ( v--v)`
Convert angle expressed in vulgar degrees to angle expressed in vulgar radians.

`: RAD>DEG ( v--v)`
Convert angle expressed in vulgar radians to angle expressed in vulgar degrees.

`variable 'D~`
State variable for `D~`.

`: D~ ( d1 d2--d3)`
`D~` alternately adds d1 to d2 or subtracts d2 from d1, giving result d3 on successive calls.

`2variable X^2`
Used by `UFSIN` and `UFCOS` as a multiplier for the recurrence.

`variable SIN-LOOPS`
Used by `UFSIN` to control the maximum number of iterations, for optimisation purposes. Values are implementation dependent.

`: UFSIN ( uf--uf)`
Returns sine of argument in scaled format as above. Argument is also unsigned scaled double, in radians. Valid range is 0 to 45 degrees, later reduced to 0 to 1 degree.

`variable COS-LOOPS`
Used by `UFCOS` to control the maximum number of iterations, for optimisation purposes. Values are implementation dependent.

`: UFSIN ( uf--uf)`
Returns cosine of argument in scaled format as above. Argument is also unsigned scaled double, in radians. Valid range is 0 to 45 degrees, later reduced to 0 to 1 degree.

`: MAKE-TABLE ( )`
Constructs parameter field of `SIN-TABLE`.

`create SIN-TABLE`
Holds sines of angles 0 to 90 degrees in degrees as scaled doubles, as above.

`: SIN@ ( n--uf)`
Returns sine of integer n, range 0 to 90 degrees, in scaled format.

`variable -SIN?`
*True* if argument most recently returned by `INT-SIN` should be negative, *false* otherwise.

`: INT-SIN ( n--uf)`
Returns sine of integer n, range 0 to 360 degrees, in scaled format.

`variable -COS?`
*True* if argument most recently returned by `INT-COS` should be negative, *false* otherwise.

`: INT-COS ( n--uf)`
Returns cosine of integer n, range 0 to 360 degrees, in scaled format.

`2variable COSI`
Used by `VSIN-COS`. Holds cosine of integer part of argument.

`2variable COSF`
Used by `VSIN-COS`. Holds cosine of fractional part of argument.

`2variable SINI`
Used by `VSIN-COS`. Holds sine of integer part of argument.

`2variable SINF`
Used by `VSIN-COS`. Holds sine of fractional part of argument.

`: STRIP.SIGN ( v--f v)`
Returns absolute value of vulgar. Flag is *true* if vulgar was negative, *false* otherwise.

`: PARTIAL.RESULTS ( v)`
Sets variables used by `VSINCOS`, based on argument, a positive vulgar representing an angle in degrees.

`: DO-SIN ( f--v)`
Returns sine of argument to `VSINCOS`, as a vulgar. Result is negated if f is *true*, i.e., argument to `VSINCOS` was negative.

`: DO.COS ( --v)`
Returns cosine of argument to `VSINCOS`, as a vulgar.

`: VSINCOS ( v--v1 v2)`
Returns sine (v1) and cosine (v2) of argument, an angle in degrees, expressed as a vulgar.

`: VSIN ( v--v)`
Returns vulgar sine of vulgar argument, in degrees.

`: VCOS ( v--v)`
Returns vulgar cosine of vulgar argument, in degrees.

`: VTAN ( v--v)`
Returns vulgar tangent of vulgar argument, in degrees.

`: VSEC ( v--v)`
Returns vulgar secant of vulgar argument, in degrees.

`: VCOSEC ( v--v)`
Returns vulgar cosecant of vulgar argument, in degrees.

`: VCOT ( v--v)`
Returns vulgar cotangent of vulgar argument, in degrees.

`: >> ( u--u)`
Returns top two bits of number, shifted to extreme right.

`: 4U/MOD ( u1--u2 u3)`
u3 is result of dividing unsigned number u1 by four; u2 is the remainder. Uses bit-masking for speed.

`: 4Q/MOD ( uq1--u uq2)`
uq2 is the unsigned quad result of dividing unsigned quad uq1 by four; u is the remainder. Uses bit-masking for speed.

`: Q0= ( q--f)`
Flag is *true* if quad argument is zero, *false* otherwise.

`: D2* ( d--d)`
Doubles double number.

`: D4* ( d--d)`
Multiplies double number by four.

`: (QSQRT) ( u uq--ud ud)`
Recursive portion of QSQRT.

`: QSQRT ( uq--ud)`
Returns double square root of quad number. Result may be in error if uppermost two bits of uq are 1.

`: >TOP ( v--v)`
Denormalises vulgar number by shifting numerator and denominator as far left as possible without altering the value of v.

`: VSQRT ( v--v)`
Returns vulgar square root of vulgar number.

`: DNOT ( d--d)`
Returns bit-wise inverse of double number.

`: DXOR ( d1 d2--d)`
Returns bit-wise exclusive-or of doubles d1 and d2.

`: FUNC-G ( d dc1 dc2--d)`
Weird. See *Numerical Recipes* for details.

`: PSEUDO-DES ( d d--d d)`
Weird. See *Numerical Recipes* for details. For 16-bit version, omit portion of double literals after the comma. In ANS systems, the comma should be a period.

`2variable COUNTER`
State variable for RANDOM.

`2variable SEQUENCE#`
State variable for RANDOM.

`: START-SEQUENCE ( d d)`
Initialises random-number generator. TOS specifies which random sequence is to be generated. All double numbers specify different sequences. 2OS specifies where in the sequence to commence. All double numbers specify different starting positions.

`: RANDOM ( --d)`
Returns the next double number from the specified sequence of random numbers. All numbers are equally likely.

`: VRAND ( --v)`
Returns a random vulgar in the range $0 \le v \le 1$. Sequences are as per RANDOM.

***MAKE YOUR SMALL COMPUTER***
***THINK BIG***
(We've been doing it since 1977 for IBM PC, XT, AT, PS2, and TRS-80 models 1, 3, 4 & 4P.)

**FOR THE OFFICE** — Simplify and speed your work with our outstanding word processing, database handlers, and general ledger software. They are easy to use, powerful, with executive-look print-outs, reasonable site license costs and comfortable, reliable support. Ralph K. Andrist, author/historian, says: "**FORTHWRITE** lets me concentrate on my manuscript, not the computer." Stewart Johnson, Boston Mailing Co., says: "We use **DATAHANDLER-PLUS** because it's the best we've seen."

**MMSFORTH System Disk** from $179.95
**Modular pricing** — Integrate with System Disk only what you need:

| | |
|---|---|
| **FORTHWRITE** - Wordprocessor | $99.95 |
| **DATAHANDLER** - Database | $59.95 |
| **DATAHANDLER-PLUS** - Database | $99.95 |
| **FORTHCOM** - for Communications | $49.95 |
| **GENERAL LEDGER** - Accounting System | $250.00 |

***MMSFORTH***

**MILLER MICROCOMPUTER SERVICES**
61 Lake Shore Road, Natick, MA 01760
(508/653-6136, 9 am - 9 pm)

**FOR PROGRAMMERS** — Build programs FASTER and SMALLER with our "Intelligent" MMSFORTH System and applications modules, plus the famous MMSFORTH continuing support. Most modules include source code. Ferren MacIntyre, oceanographer, says: "Forth is the language that microcomputers were invented to run."

**SOFTWARE MANUFACTURERS** — Efficient software tools save time and money. MMSFORTH's flexibility, compactness and speed have resulted in better products in less time for a wide range of software developers including Ashton-Tate, Excalibur Technologies, Lindbergh Systems, Lockheed Missile and Space Division, and NASA-Goddard.

**MMSFORTH V2.4 System Disk** from $179.95
Needs only 24K RAM compared to 100K for BASIC, C, Pascal and others. Convert your computer into a Forth virtual machine with sophisticated Forth editor and related tools. This can result in 4 to 10 times greater productivity.

**Modular pricing** — Integrate with System Disk only what you need:

| | |
|---|---|
| **EXPERT-2** - Expert System Development | $69.95 |
| **FORTHCOM** - Flexible data transfer | $49.95 |
| **UTILITIES** - Graphics, 8087 support and other facilities. | |

***and a little more!***

**THIRTY-DAY FREE OFFER** — **Free MMSFORTH GAMES DISK** worth $39.95, with purchase of MMSFORTH System. CRYPTOQUOTE HELPER, OTHELLO, BREAKFORTH and others.

***Call for free brochure, technical info or pricing details.***

by hex 4000 after >TOP is applied in a 16-bit system.

## Vulgar Random Numbers

As I have no idea what REDUCE would do to a number generated by, say, a linear congruential generator, I have assumed that it is probably death to anything but the most robust random-number generators. Therefore, I have used the best one I could find, which is RAN4 from the second edition of *Numerical Recipes* by Press, et al. This has been extended using the method they advise to generate 64-bit random numbers. (16 bitters, omit the part of the double number that comes after the comma. This gives 32-bit random numbers on a 16-bit system. My system uses commas, not periods, in double numbers.)

To start up, the generator takes two double numbers. The top one selects which sequence will be generated, and the second indicates where in the sequence to start from. RANDOM then spits out successive numbers in that sequence. All the bits are good, so the range can be reduced using MOD with impunity. How RANDOM works is very technical; see *Numerical Recipes* and be prepared to have your thinking gland put through the wringer.

VRAND returns uniform deviates in the range $0 \le V \le 1$. (Rounding is the nature of the beast; if you prefer $0<V<1$, test explicitly for zero and one and call RANDOM again.)

## Conclusion

This completes, at least for the time being, my exposition of rational number representations. I would be pleased to hear from anyone who cares to define additional higher functions, or who can suggest improvements to the code given here.

## References

In addition to the references given in the previous article, I mention the *Longman Mathematics Handbook* by Keith Selkirk, which now falls open on the pages about sine and cosine if I even glance in its general direction.

Good books about Logo and turtle graphics are *Computer Science Logo Style* by Brian Harvey, and *Turtle Geometry* by Harold Abelson and Anrea diSessa.

Doing square roots longhand is clearly explained in the *Encyclopaedia Britannica.*

Gordon Charlton was named after Charlton Heston. He has been described as tall, handsome, witty, intelligent, charming, and world famous. Gordon is also quite tall. He wishes to thank readers of *Forth Dimensions* for not responding to his request for a rigorous description of the Ratcliffe-Obershelp algorithm, as this prompted him to develop a rather better one himself.

**Listing: Vulgar Functions**

```
\ unsigned double-length fractional multiplication
: T* ( ud u--ut) tuck um*  2swap um*  swap >r 0 d+  r> -rot ;

: D= ( d d--f)  rot =  -rot =  and ;

: Q* ( ud ud--uq) swap 2over rot  >r >r >r  t*
                  rot  r> r> r> t*  rot >r  rot 0 d+
                  swap >r 0 d+  r> r> swap 2swap ;

: Q>UF ( uq--uf) 2over  0 highbit d=
                 IF 2swap 2drop over 1 and
               ELSE 2swap 0 highbit ud< not negate THEN 0 d+ ;

: UF* ( uf uf--uf)  q* q>uf ;

\ UD-fractional conversion routines
: MU/R ( ud u--ud) >r 0 r@ um/mod  r@ swap >r um/mod  r>
         rot 2* dup r@ =
         IF r> 2drop dup 1 and  ELSE  r> u> negate THEN  0 d+ ;

: T/ ( ut u--ud) dup >r um/mod -rot r> um/mod nip swap ;
: M*/ ( ud u u--ud) >r t* r> t/ ;

-1, 2constant ONE
: UF>V ( uf--v) 2 mu/r [ one 2 mu/r ] dliteral reduce ;
: V>UF ( v--uf) one 2swap m*/ ;
```

```
3,141592653589793238 vulgarise 2constant PI
: DEG>RAD ( v--v) [ pi 180 s>v v/ ] dliteral v* ;
: RAD>DEG ( v--v) [ 180 s>v pi v/ ] dliteral v* ;

\ UD-fractional sine, cosine (by Taylors series expansion)
variable 'd~
: D~ ( d d--d) 'd~ @ dup not 'd~ ! IF d+ ELSE d- THEN ;

2variable X^2
variable SIN-LOOPS 10 sin-loops !
: UFSIN  ( uf--uf) 0 'd~ ! 2dup 2dup 2dup uf* x^2 2!
     sin-loops @ 1 DO  x^2 2@  i 2* dup 1+ * mu/r  uf*
                       2dup or 0= IF  LEAVE  THEN
                       2dup >r >r  d~  r> r>  LOOP  2drop ;

variable COS-LOOPS  10 cos-loops !
: UFCOS  ( uf--uf) 0 'd~ !  2dup uf*  x^2 2!  one one
     cos-loops @ 1 DO  x^2 2@  i 2* dup 1- * mu/r  uf*
                       2dup or 0= IF  LEAVE  THEN
                       2dup >r >r  d~  r> r>  LOOP  2drop ;

\ integer-fractional sine and cosine (by table lookup)
: MAKE-TABLE  46 0 DO  i s>v deg>rad v>uf ufsin , ,      LOOP
              0 44 DO  i s>v deg>rad v>uf ufcos , ,  -1 +LOOP ;

create SIN-TABLE   make-table
: SIN@ ( n--uf) 2* cells sin-table + 2@ ;

variable -sin?
: INT-SIN ( n--uf) dup 189 > dup -sin? ! IF 180 - THEN
                   dup  90 > IF 180 swap - THEN sin@ ;

variable -cos?
: INT-COS ( n--uf) dup 180 > IF 360 swap - THEN
      dup  89 > dup -cos? ! IF 90 - ELSE 90 swap - THEN sin@ ;

\ vulgar sine-cosine (by serial expansion/table hybrid)...
3 sin-loops !  4 cos-loops !  \ UFSIN, UFCOS limited to 0-1 deg

2variable COSI  2variable COSF  2variable SINI  2variable SINF

: STRIP.SIGN ( v--f v)    2dup v0< -rot vabs ;

: PARTIAL.RESULTS ( v)
   2dup v>s 360 mod    dup int-sin sini 2!  int-cos cosi 2!
   vfrac deg>rad v>uf  2dup  ufsin sinf 2!    ufcos cosf 2! ;

: DO.SIN ( f--v) sini 2@ cosf 2@ uf*  cosi 2@ sinf 2@ uf*
                 -sin? @  -cos? @ =  IF d+ ELSE d- THEN
                 uf>v  rot -sin? @ xor IF vnegate THEN ;

\ vulgar sine-cosine (by serial expansion/table hybrid)
: DO.COS ( --v)   cosi 2@  cosf 2@ uf*  sini 2@  sinf 2@ uf*
                  -sin? @  -cos? @ =  IF d- ELSE d+ THEN
                  uf>v  -cos? @ IF vnegate THEN ;

: VSINCOS ( v--v v) strip.sign partial.results do.sin do.cos ;
: VSIN ( v--v)  vsincos 2drop ;
: VCOS ( v--v)  vsincos 2swap 2drop ;
: VTAN ( v--v)  vsincos v/ ;

: VSEC   ( v--v) vcos reciprocal ;
: VCOSEC ( v--v) vsin reciprocal ;
: VCOT   ( v--v) vtan reciprocal ;

\ quad square root -- low-level bit manipulation
: >> ( u--u) dup 1 and IF [ highbit 2/ highbit xor ] literal
                    ELSE 0 THEN
            swap 2 and IF highbit or THEN ;

: 4U/MOD ( u--u u) dup 3 and
                 swap 2/ 2/ [ highbit 2/ not ] literal and ;

: 4Q/MOD ( q--u q) 4u/mod >r  >r 4u/mod r> >> or >r
      >r 4u/mod r> >> or >r  >r 4u/mod r> >> or r> r> r> ;

: Q0= ( q--f) or or or 0= ;

: D2* ( d--d) 2dup d+ ;
: D4* ( d--d) d2* d2* ;

\ quad and vulgar square root
: (QSQRT) ( u q--ud ud) 4dup q0=
          IF  2drop  0 -rot
        ELSE  4q/mod  RECURSE
              d2* 2dup >r >r
              d2*  1, d+  4dup  ud<
              IF  2drop d4*  0 -rot d+  r> r>
            ELSE  d- d4*  0 -rot d+  r> r> 1, d+ THEN THEN ;

: QSQRT ( uq--ud) >r >r  0 -rot r> r> (qsqrt) 2swap 2drop ;

: >TOP ( v--v) BEGIN  2dup or highbit 2/ and 0=
               WHILE  2* swap 2* swap  REPEAT ;

: VSQRT ( +v--v) >top dup >r um* 0 0 2swap qsqrt 0 r> reduce ;

\ vulgar random-number generator -- 32-bit Version
: DNOT ( d--d)  swap not  swap not ;                     hex
: DXOR ( d d--d) rot xor >r xor r> ;
: FUNC-G ( d dc1 dc2--d) >r >r dxor 2dup um* 2swap dup um*
                     dnot rot dup um* d+ swap r> r> dxor d+ ;

: PSEUDO-DES ( d d--d d)
  2swap 2over BAA96887,E34C383B 4B0F3B58,3D02B5F8 func-g dxor
  2swap 2over 1E17D32C,39F74033 E874F0C3,9226BF1A func-g dxor
  2swap 2over 03BCDC3C,60B43DA7 6955C5A6,1D38CD47 func-g dxor
  2swap 2over 0F33D1B2,65E9215B 55A7CA46,F358B432 func-g dxor ;
2variable COUNTER  2variable SEQUENCE#
: START-SEQUENCE ( dcounter dseq#) sequence# 2! counter 2! ;
: RANDOM ( --d) sequence# 2@  counter 2@  pseudo-des
          2swap 2drop counter 2@ 1, d+ counter 2! ;
: VRAND ( --v) random highbit not and
              [ -1 -1 highbit xor ] dliteral reduce ;   decimal
```

F83

# Convert Real Numbers to Fractions

Walter J. Rottenkolber
Mariposa, California

Most formulas use real numbers, especially in the odd constant, logarithm, or trigonometric value. Charles Moore prefers Forth to use scaled integer arithmetic. So how do you find the fraction that best describes the real number, especially if the fraction needs to be small enough to use in signed number operations?

This program generates a list of fractions equivalent to a real number by means of a mathematical concept called *continued fractions*. This is based on Euclid's algorithm for the greatest common denominator, and Knuth explores it in depth. A more popular description for C mavens was given by Mark Gingrich.

A short version of pi = 3.1416 could be represented by a continued fraction:

$$3 + \cfrac{1}{7 + \cfrac{1}{16 + \cfrac{1}{11 + \dots}}}$$

Mathematicians use the shorthand [3;7,16,11,...] which eliminates the denominator of 1 for the initial integer, and the numerator of 1 for the fractions.

This fraction series can be rewritten as:

```
3/1 + 1/7 + 1/16 + 1/11 ...
```

If the real number is rational, the series will stop; but if the number is irrational, as most are, the series continues indefinitely. As you add up the fractions, the real number generated by dividing through the summed fraction will oscillate about the true value of the real number while steadily approaching it. In the above series of fractions, the first three add up to 355/113, which approximates pi to six decimal places.

The program generates such a fraction series, and sums it at each step into a single fraction so that a list of fractions is created.

To run the program, you must first convert the real number to a fraction to the base (radix) ten. If you start with pi = 3.1416, convert it first to a fraction by dividing by an appropriate multiple of ten, i.e., 31416/10000.

Run the program by typing `31416 10000 FRACLIST`

As a first approximation, we have 3 + (1416/10000).

The remainder is then processed as 1/1/(1416/10000) or 1/(10000/1416) or 1/7 + (88/1416). So, pi now is 3 + 1/7 + (88/1416) or 22/7 + (88/1416).

This remainder can be further processed as 1/1/(88/1416) or 1/(1416/88) or 1/16 + (8/88). So pi now is 22/7 + 1/16 + (8/88) or 355/113 + (8/88).

The last step is 355/113 + 1/1/(8/88) or 355/133 + 1/(88/8) or 355/113 + 1/11. Now, there is no remainder, so we should have the original fraction or its equivalent. Calculating it out: ( 355 11 * 22 + ) / ( 113 11 * 7 + ) or 3927/1250 = 3.1416, and the process ends.

Usually, the larger fractions are more accurate, but not necessarily, so you need to check. In the above list, 355/113 = 3.1415929, and 3927/1250 = 3.1416. Since pi = 3.14159265, you can see that the smaller fraction is actually the more accurate.

Running the program with pi = 3.1415 will produce a longer and somewhat different list.

The precision and range of the program depends on the size of the integers used (i.e., 16- or 32-bit integers) and whether unsigned integers are used. It's important that you double-check the value of the real number generated by the fraction (I use a pocket calculator), and that the size of the numbers in the fraction match the range of the scaling operators (e.g., */) used in the calculation. Remember, most scaling operators use signed integers.

Two programs are listed. The first is based on unsigned 16-bit integers, because Laxen and Perry's F83 has limited double-arithmetic routines. This restricts the range of real numbers that can be processed, though accuracy can be surprisingly good. The second program can be used if your Forth has a double, unsigned multiply and divide. The double-integer routines of Tim Hendtlass work just fine.*

## References

Knuth, Donald E. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms,* 2nd ed., pg. 339. Reading, Massachusetts: Addison-Wesley.

Gingrich, Mark. *The C Users Journal,* Feb. 1993. Vol. 11, no. 2, pp. 35–43.

Hendtlass, Tim. "Math—Who Needs It?" *Forth Dimensions,* Vol. 14, no. 6.

*In Tim Hendtlass' 32-bit routines ("Math—Who Needs It?" *FD* XIV/6), he uses some words that do not grace Laxen and Perry's F83. These are best handled by in-line expansions as follows:

```
2>R     =>   >R >R
2R>     =>   R> R>
2R@     =>   R> R@ OVER >R
DUP>R   =>   DUP >R
```

If you attempt to define a substitute word, e.g.,

```
: 2R>  ( — n n ) R> R> ;
```

you will only mess up the return stack and cause a system crash.

Walter J. Rottenkolber bought his first computer in 1983. Early on, he experimented with fig-Forth and other languages, but gravitated to assembler until reintroduced to Forth in 1988. Forth provides the same close-to-the-silicon feeling as assembler, but without the pain. Interests include small embedded systems, programming, and computer history, about which he enjoys writing.

```
    11
\ Real Number to Fraction Generator                WJR01MAR93

VARIABLE OLDNUM       VARIABLE NEWNUM
VARIABLE OLDDENOM     VARIABLE NEWDENOM

: U*     ( u u -- u ) UM* DROP ;
: U/MOD  ( u u -- ur uq ) 0 SWAP UM/MOD ;
: U/     ( u u -- uq )  U/MOD NIP ;
: UMOD   ( u u -- ur )  U/MOD DROP ;

: INITFRAC  ( -- )
    0 OLDNUM !      1 NEWNUM !
    1 OLDDENOM !    0 NEWDENOM ! ;

-->

```

```
    12
\ Real Number to Fraction Generator                WJR13MAR93

: SAVNUM    ( -- )  NEWNUM @ OLDNUM ! ;
: SAVDENOM  ( -- )  NEWDENOM @ OLDDENOM ! ;

: CALCNUM   ( u -- )
   NEWNUM @ U* OLDNUM @ + SAVNUM NEWNUM ! ;

: CALCDENOM  ( u -- )
   NEWDENOM @ U* OLDDENOM @ + SAVDENOM NEWDENOM ! ;

: PRNFRAC  ( -- )
   NEWNUM @ IF CR NEWNUM @ 8 U.R
   ."  / " NEWDENOM @ U. THEN ;

-->
```

```
    13
\ Real Number to Fraction Generator                WJR15MAR93

: NXTFRAC  ( num denom -- num' denom' )  TUCK UMOD ;
: ENDFRAC?  ( num denom -- f )  UMOD 0= ;

: CALCFRAC  ( num denom -- )
   2DUP U/ DUP CALCNUM
   CALCDENOM  PRNFRAC NXTFRAC  ;

: LASTFRAC  ( u u -- )  CALCFRAC 2DROP ."   Done" CR ;

: FRACLIST  ( num denom -- )
   INITFRAC
   BEGIN  CALCFRAC 2DUP ENDFRAC? UNTIL
   LASTFRAC ;

```

```
    14
\ Real Number to Double Fraction Generator          WJR15MAR93

2VARIABLE OLDNUM       2VARIABLE NEWNUM
2VARIABLE OLDDENOM     2VARIABLE NEWDENOM


: INITFRAC  ( -- )
    0. OLDNUM 2!      1. NEWNUM 2!
    1. OLDDENOM 2!    0. NEWDENOM 2! ;

-->
```

```
    15
\ Real Number to Double Fraction Generator          WJR15MAR93

: SAVNUM    ( -- )  NEWNUM 2@ OLDNUM 2! ;
: SAVDENOM  ( -- )  NEWDENOM 2@ OLDDENOM 2! ;

: CALCNUM   ( ud -- )
   NEWNUM 2@ UD* OLDNUM 2@ D+ SAVNUM NEWNUM 2! ;

: CALCDENOM  ( ud -- )
   NEWDENOM 2@ UD* OLDDENOM 2@ D+ SAVDENOM NEWDENOM 2! ;

: PRNFRAC  ( -- )
   NEWNUM 2@ D0= NOT IF CR NEWNUM 2@ 11 UD.R
   ."  / " NEWDENOM 2@ UD. THEN ;

-->
```

```
    16
\ Real Number to Double Fraction Generator          WJR15MAR93

: NXTFRAC  ( ud-num ud-denom -- ud-num' ud-denom' )
   2SWAP 2OVER UDMOD ;

: ENDFRAC?  ( ud-num ud-denom -- f )  UDMOD D0= ;

: CALCFRAC  ( ud-num ud-denom -- )
   4DUP UD/ 2DUP CALCNUM
   CALCDENOM  PRNFRAC NXTFRAC  ;

: LASTFRAC  ( ud ud -- )  CALCFRAC 2DROP 2DROP ."   Done" CR ;

: DFRACLIST  ( ud-num ud-denom -- )
   INITFRAC  BEGIN  CALCFRAC 4DUP ENDFRAC? UNTIL
   LASTFRAC ;
```

Forth 83 Model

*(Letters, from page 6.)*

has been used by many to develop applications and perform system integration without having to deal with the complexity of a full-blown UNIX environment up front. Traces of this Forth heritage still exist in the boot process on most SPARC-based UNIX workstations. Today we see that this heritage is being extended to other systems as well, in the form of Open Firmware.

Open Firmware is an outgrowth of the IEEE P1275 working group on boot firmware which promises to provide a high degree of processor and peripheral independence. The goal is that the processor will not have to have prior knowledge about which specific peripherals happen to be connected. During the boot process, all peripherals will be probed to identify themselves and to provide their software drivers as needed. True independence is achieved by using Fcode, a kind of Forth, both in the processor's boot process and in the peripheral's BIOS. The processor gains the advantage of not having to be aware of all the various kinds and models of peripherals that might possibly be connected. Instead, each peripheral is responsible for telling the processor what it is and how to communicate with it. The peripheral gains the advantage of being able to specify a common interface and software driver regardless of which specific processor it is connected to. Once probed, it provides a universal software driver (written in Fcode) to the processor which is interpreted or compiled to establish the communication link. The communication is two-way, so if there happens to be an advantage to the processor or the peripheral to use other than this Fcode interface, that information can be passed on and utilized to establish a specific link. A good, one-page description of this is found in the April 1994 issue of *BYTE* (page 63).

Also, I have found Forth in high use at the other end of the computing spectrum, application-specific languages and environments. FIG offers a small book titled *Write Your Own Programming Language Using C++* which gives a taste of this, but the one premier example I found of this is TILE.

TILE is a Forth written in C and intended for use in the UNIX environment. It is not a toy language nor is it just an educational example. (This is *not* to imply that other Forths are toys.) Rather, it allows interactive development of applications without continuous recompilation and without exhibiting the performance degradation that plagues most interpretive programming environments.

As the application environment develops, selected pieces of code can be compiled as new Forth primitives and/or re-coded and compiled in C or another language of choice. This is not new to Forth. We have always allowed recompilation of the kernel to take advantage of the speed of compiled code. What makes TILE interesting is the ease in which this is done. TILE also allows dynamic linking with compiled modules from other languages, and in this sense it maximizes code reuse.

Also note that, by building on C, no special handles or modifications of the environment commonly found in UNIX systems had to be generated. Common UNIX tools like profiles, linkers, editors, and such are all still available. Yet while running in TILE you have the look and feel of a full-blown Forth development environment. If you desire to explore the use of Forth or application-specific languages under UNIX, you should take a look at TILE. (By the way, early versions of TILE were easily ported to DOS.)

As a Forth enthusiast, I am glad to see these developments. I feel lucky that my job allows me to work at both ends of the computing spectrum and that I can use a common programming tool throughout: Forth.

Warren Bean
Austin, Texas

### Forth's Three Problems

Dear Mr. Ouverson:

Before I launch into what will be a very critical letter, let me say that nothing I say should be taken personally, by you or by any of the many people who have brought Forth and FIG to where they are today. Your contributions are all selfless and noteworthy. To introduce myself, I am not a professional programmer, though I was a project engineer on a vacuum tube computer in 1949–1952, and have programmed professionally in the distant past and so am not without some perspective.

However, I am disappointed! I recently rejoined FIG after an absence of some four years. What I find is no evidence at all that the four years have led to any progress. Quite the opposite, when I left I had the feeling that the then-new progress toward an ANSI standard would move Forth into the mainstream and make of it the serious programming system I was looking for. What I find now is that the ANSI standard is not listed or even mentioned in the FIG publications insert and appears to have been relegated to only occasional reference here and there in *FD*.

Before going on, let me say amen to Jim Mack's letter in *FD* (XV/5). It had the effect of inspiring me to write a similar, if somewhat narrower, opinion. Meuris, Vande Keere, and Vandewege also made some telling, if less emotional, points. Their thrust toward embedded real-time control is of course valid, but if that is all Forth can be then it is not for me!

Forth has only three problems: no effective standard, no effective standard, and no effective standard. ANSI may exist (I cannot even tell from *FD*), but if FIG does not support it strongly, its existence is academic! What do I suggest? In a word, leadership!

Clearly, the creative thrust that has taken Forth so far in so many directions must not be blunted. Rather, a common origin for that thrust must be established. In what follows, I use the ANSI standard as a target. I recognize my own limited knowledge in this respect and would readily accept the substitution of a different standard if those more knowledgeable than myself can agree upon it. Some, not which, standard is what is important, but widespread acceptance is essential.

1. FIG must make the ANSI standard its centerpiece. If the standard exists, copies must be made available to all FIG members at nominal or no cost. If the standard

does not yet exist, the main focus of *FD* should be to bring it into existence quickly and spread the good news.

2. *FD* should publish *no* code that is not ANSI standard, or made to be ANSI standard by an appropriate and required prefix. No exceptions!
3. Advertisements from Forth vendors must be required to state clearly the status of all products advertised regarding the ANSI standard, and further to offer ANSI standard upgrades if they do not meet the standard.
4. The creation of a low-cost basic ANSI standard Forth system must have high priority. We had fig-Forth before, what we need now is ANSI fig-Forth (FIGA-Forth?) for all of the popular platforms.
5. All products distributed by FIG must meet the same requirements placed on advertisers.

The above is probably not sufficiently complete, but I hope it conveys the message—namely, Leadership!

For myself, I can only say that unless I see strong leadership emerging, I must once again abandon the ship. The current version of FIG simply does not deserve support! I love Forth, but it is dying and crying out for leadership!

With great hope,
Bernard H. Geyer
Prescott, Arizona

*Editor's reply:*
*As a preface, I should note that creation of an ANS language standard usually requires years of collaboration between interested parties who follow a process that is strictly controlled and supervised by ANSI. The lengthy process appears to be designed to ensure access, completeness, care, and fairness, and to prevent special interests or factions from co-opting any standard. (The mill grinds slowly but, it is hoped, very finely.) I'm just the editor around here, not the apologist or policy maker; but I will respond briefly to the above points and let others take more time to distill their thinking onto paper or diskette and send it to* FD *for the next issue(s).*

**We strongly advise professional Forth programmers to take advantage of a product from one of the Forth vendors.**

1. *FIG President John Hall says FIG does support ANS Forth. And while ANSI controls the publication and distribution of the official standard document, FIG has been working on a quick-reference guide to ANS Forth which may (if ready in time) be included free to members with this issue.*
2. *This point has been debated whenever a new Forth standard has been published. It undoubtedly will be debated again in the case of ANS Forth. I will just say that, while this seemingly small step would put a tidy organizational seal of approval on ANS Forth, it also would greatly diminish* FD*'s support of readers who choose not to adopt the new standard immediately or who are not yet skilled enough in Forth to translate from one dialect to another, and it would censor some valuable authors whose work is not written in the current standard of choice.*
3. *I wouldn't want to* require *advertisers to do this, but smart vendors of ANS-compliant products certainly should trumpet that news in* FD *advertising. That's basic business sense. (Right?)*
4. *You really lit the fuse with this one. Vendors historically have complained that FIG's distribution of*any *low-cost or public-domain Forth system hurts their sales. On the other hand, such systems' use as outreach tools, introductions to Forth, and platforms for experimentation and innovation probably provide some benefits to everyone in the Forth business. On the other, other hand... the arguments go on and on.*

   *As John Hall recently wrote in a press release regarding ANS Forth, "For serious development, we strongly advise professional Forth programmers to take advantage of a product from one of the Forth vendors. Those vendors can offer the technical support, bug fixes, and stability that software developers cannot afford to be without..." I wonder how the incidental sales of public-domain Forths can compete in any serious way with professional systems. There is no tech support, no R&D budget, little or no documentation, no consulting service, and no marketing. Someone will undoubtedly provide a public-domain ANS Forth, if history repeats itself, but surely that would not have any real impact on those business-minded vendors who create, market, and support the professional-quality systems that professional-quality programmers and developers should be buying and using.*
5. *I wore out my typing fingers on the last point, and will simply pass your comment along to FIG management for consideration...*

*Disclaimer: I do not attempt speak for ANSI, not even for the X3J14 technical committee that formulated ANS Forth. I encourage X3J14 participants—collectively or individually—and others knowledgeable about ANS Forth to write articles or letters to* Forth Dimensions *about the standard, the standardization process, or specific issues such as those raised in the Mr. Geyer's letter.*

*—M.D.O.*

*X3J14 Chair replies:*
Dear Marlin:

I heartily sympathize with Mr. Geyer's amazement and frustration at the excruciatingly long time it has taken for ANS Forth to become final. For the last two years we have made only fine-tuning corrections and clarified explanatory text; the last actual technical change was voted in

January, 1993. ANSI's machinery grinds slowly. We received notice only recently of their final official approval, which occurred March 26, 1994.

As you note, publication of the standard is controlled by ANSI, and it is not cheap: drafts were priced at $60, and I doubt the final standard will be any cheaper. This is typical of prices charged for language standards. X3J14 has explored the feasibility of electronic publication, but have not received a definitive answer. Meanwhile, copies of a draft published in August, 1993, for "typo editing" are still available on most Forth BBSs (GEnie, CompuServe, Internet, etc.). Differences between this document and the official standard are typographical only.

The official standard bears the designation ANSI X3.215.1994, *Programming Language Forth*. It will be available "soon" from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704, 800-854-7179, or fax 303-843-9880.

As you note, vendor compliance both in their products and advertising will respond to market pressure. We have evolved our systems toward the standard, and are compliant in most respects. Most vendors I know are working on fully compliant versions, but it may take a while. We, like others, have many thousands of lines of code to be evaluated, possibly modified and re-tested, not to mention extensive, expensive documentation changes. Our customers have *millions* of lines of code, and would be justifiably outraged if we switched overnight, even assuming we could. Compatibility shells will help, and some compliant PD systems are emerging.

The suggestion that *FD* set a requirement for published code to be ANS compatible is a good one. A similar requirement was instituted by (then) editor Leo Brodie with respect to Forth-83. It shouldn't be too hard for most authors to comply. ANS Forth can immediately serve to unite the community beyond the "Tower of Babel" of dialects by providing a common communication medium.

Sincerely yours,
Elizabeth D. Rather, President
FORTH, Inc.
111 N. Sepulveda Blvd.
Manhattan Beach, California 90266-6847

**Interfacing with Electric Dreams**

Dear Marlin,

"Please cancel my subscription" is normally all I would write for a letter of cancellation, but *Forth Dimensions* represents more to me than just a magazine. It represents the way I believe computers ought to be programmed. However, the way computers ought to be programmed is no longer my focus.

My focus is and has been computer animation. I believed that the control and interactivity of Forth would allow me to create animation tools. I saw other people do similar things, but they did not provide what I needed and they did not provide an interface that lent itself to artistic spontaneity. I found this spontaneity and control in SuperCard, HyperCard, MetaCard, ToolBox, and ScriptX.

I then tried to design something like SuperCard in Forth while using SuperCard to do animation. This proved to be difficult because I could not find a job or sell a product that allowed me to pursue either of these directions. I then tried to cram both pursuits into the little time I had after trying to make a living. With this conflicted and small amount of time, I did not make much progress on either front.

I decided to focus my time, energy, and money on creating animation using the scripting languages. I hope that someone with more of a focus on Forth will design a "ForthCard" for me and the rest of the world.

Such an application would allow the user to have the ease of use of SuperCard, but its engine could be smaller. It would allow users to create new types of objects. It would provide non-programmers with a means of getting their own special tasks done and a way for them to become programmers. Such a friendly, object-oriented, and media-oriented product would prove that Forth is the way to program computers.

Perhaps once I can get my animation career off the ground I will re-subscribe and return to using Forth to create the scripting language I imagine. For now, thanks to *Forth Dimensions* for many years of electric dreams.

Regards,
Mark Martino
Redmond, Washington

**Total control with *LMI FORTH*™**

***For Programming Professionals: an expanding family of compatible, high-performance, compilers for microcomputers***

**For Development:**
Interactive Forth-83 Interpreter/Compilers for MS-DOS, 80386 32-bit protected mode, and Microsoft Windows™
- Editor and assembler included
- Uses standard operating system files
- 500 page manual written in plain English
- Support for graphics, floating point, native code generation

**For Applications:** Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 64180, 680X0 family, 80X86 family, 80X96/97 family, 8051/31 family, 6303, 6809, 68HC11
- No license fee or royalty for compiled applications

*Laboratory Microsystems Incorporated*
*Post Office Box 10430, Marina Del Rey, CA 90295*
*Phone Credit Card Orders to: (310) 306-7412*
*Fax: (310) 301-0761*

A Forum for Exploring Forth Issues and Promoting Forth

# Fast FORTHward

Mike Elola
San Jose, California

## Answering Leo Brodie's OBJECTions

In *Thinking Forth*, Leo Brodie criticized software layers, objects, and modules. Brodie sees lost opportunities for reuse whenever routines are made inaccessible by our placing them in the private part of a module or class. He raises a legitimate concern that the same primitive routines might be implemented repeatedly inside such modules.

Brodie praises Forth for its egalitarian treatment of low- and high-level routines. He emphasizes how Forth allows the same programming philosophy to apply at all levels of coding in statements such as "All code should look and feel the same."

Much of what Brodie says has a true ring to it. But even he wants to see a richer Forth landscape than data structures and routines. He espouses the virtues of components as an important unit to be nurtured within programs. Even though the two are very much alike, components receive his praise while modules (objects) are disparaged.

***By offering only some of the benefits of modules, Forth subtly sabotages more complete implementations...***

Moreover, components seem to exist in the eye of the beholder. They don't require adding anything to Forth. No new namespace management tools are needed, nor are additional scoping rules.

Components only require adherence to a modular programming style and philosophy. For example, Brodie lauds information-hiding as an important part of component design. But it takes the shape of a programmer discipline rather than a namespace management facility.

Accordingly, you might not see where one component ends and another begins. Unfortunately, this status-quo style of thinking can lead us to overlook the benefits offered by full-blown modularization tools.

## New Reuse Architectures

Classes may indeed hide some routines from view as Brodie warns—but it's no mistake that they do. The class designer enables reuse through a protocol and infrastructure known as inheritance.

In object-oriented languages (OOLs), reuse takes a form that is not so granular or direct as calling a shared subroutine. The OOLs treat the subroutine call hierarchy as a simplified case of reuse. A greater commitment to reuse requires accounting for data structures as well as routines. Data structuring is elevated in importance. The reuse of routines is regulated according to the data structures they use.

Even the module support in languages such as Ada and Modula-3 offer enhanced reuse provisions, such as formal mechanisms to generate generic operations. Brodie's components do not help diversify or extend reuse provisions as do OOLs and such languages as these.

To better ground this discussion, let's identify the benefits we should expect to be delivered by modularization tools. Two of them are: (1) refined control over the visibility of routines (improved namespace management facilities); and (2) improved management of the interdependencies between units of code.

Components don't deliver either of these benefits.

C libraries offer the benefit of improved management of code modules. For example, they manage and track the dependencies between routines. The C language also has facilities for verifying the proper use (call parameters and return values) of functions inside a library module.

C libraries also offer improved namespace management facilities, but in a dispersed fashion. Through the associated header file that you reference with an INCLUDE statement, an object module's symbol table becomes visible in whole or in part, depending on what was declared in the header file. So with the proper coordination of original code and header file code, routine visibility is subject to refined control, although perhaps not very straightforward control. It is also not a tamper-proof form of control from the perspective of the author of the module.

The object class provisions of a typical OOL are able to offer both of the basic benefits of modules in a more straightforward and better-regulated fashion. Likewise, the modules of Modula-3 and Ada deliver the benefits that we expect.

## Regulating Reuse

An underlying assumption of recent programming languages is that reuse is better served by more regulation. Certain kinds of code should not be made available for reuse.

The visibility of routines across modules ought to be controlled very discreetly. A distinction ought to be made between interface code that is available universally (as long as the module is loaded) and the module's private code. Besides making modules more robust and change-resistant, these different levels of visibility can help synchronize the code maintained in multiple modules.

## ANSI Forth Standard Makes Debut

As you probably have heard, the ANSI Forth standard has been approved. My personal opinion is that the caliber of this standardization work was excellent, and that we will be reaping the benefits of it for many years to come.

FIG will honor the occasion by releasing a quick reference card to all its members. Non-members can purchase the card for a small fee. (FIG plans to carry the standard document as well. Look for it in the mail-order form.)

(As the author of the quick reference card, I have tried to abbreviate a ton of information yet still do the standard justice. I urge you refer to additional study materials as well.)

FIG is distributing its own press release heralding the arrival of the standard. It will have gone out to 67 editors and technical journalists at a variety of journals by the time you read this. If you spot media coverage of ANS Forth, drop me an e-mail message telling me of its whereabouts. The objective is that FIG will be mentioned (along with contact information) as part of the coverage.

## For Vendors Only

I would appreciate your sending me an e-mail notice each time you send FIG a press release announcing a new product. The deadline for submissions is supposed to be the second week of odd months. However, with e-mail notification, I can reserve the space for you for an additional week.

*—Mike Elola*
*elolam@aol.com*

P.S. Don't miss the generous discount offer from Forth, Inc. announced on page 11!

## Product Watch

APRIL 1994

RAM Technology Systems Ltd. introduced their Smart-ICEPIC-20E in-circuit emulator and Interactive Remote Target Compiler (IRTC678). Together, these products provide a low-cost, IBM PC-based, interactive development environment for PIC microcontrollers.

The PC's parallel port can support one Smart-ICEPIC™ device. Multiple parallel ports can support the development of multiple-processor targets. Each Smart-ICEPIC has an 18-pin DIL lead. Larger microcontroller packages require an included "remote" cable. Currently, the PIC16C64, PIC16C71, and PIC16C84 are supported.

The IRTC678 offers a hypertext editor, assembler, simulator, talker, Forth-to-native-code compiler, and library compiler. Source may be compiled and simulated on the PC or programmed interactively into the PIC microcontroller and executed in real time.

**COMPANIES MENTIONED**

RAM Technology Systems Ltd.
Clump Fram Industrial Estate
Higher Shaftesbury Road
Blandford Forum
Dorset DT11 7TD
United Kingdom
Fax: (0258) 456410

Careful programming style is still needed to make the best use of objects or other modularization tools. For example, if some general-purpose code creeps unnoticed into a module, it should be identified and moved to another location where its reuse is not restricted. The original module can include a statement of dependency for the outlying code, if necessary (if it has been placed in yet another module, as it probably should be).

## Other Modularization Benefits

Improved code management facilities accompany most module implementations. The organization of code into modules helps make possible operations such as compiling modules, loading and running modules, and releasing modules.

If a module is already loaded, subsequent requests to load it can be automatically suppressed. Unfortunately, most Forth module implementations seize upon this as the sole purpose of modules, and they offer no other benefits, not even the basic ones I already enumerated.

(In FORML proceedings, I have found proposals that address the issues of public and private code visibility, module relocatability, intermodule communication, and even paged memory-mapped modules. John James has written a couple of these papers, and Stephen Pelc and Neil Smith wrote another.)

In any case, neither complete nor minimal implementations of modules have earned much popularity in the Forth community. Perhaps Forth impedes the acceptance of new programming practices. I'll hazard a guess that our much-admired Forth programming freedoms predispose us to dislike the extra regulation imposed by a module mechanism.

Forth usually offers some ways to control routine visibility, such as vocabularies, same-name words in one vocabulary, name-stripping, and unlinking of words from the dictionary.

Vocabularies can even be considered a germinal form of module mechanism—one that controls routine visibility. Vocabularies are one way that Forth has achieved at least a minimal level of parity with modern languages. (In the next installment, I'll explore how vocabularies are superior in certain ways to equivalent, but lexically delimited, modules.)

However, by offering some of the benefits without a full realization of all of the benefits of modules, Forth subtly sabotages more complete and formal implementations of the same.

*(Continued on page 37.)*

## The European Forth Conference, EuroForth'94
## November 4-6, 1994
## *Exploiting Forth: Professionally, Commercially, & Industrially*

EuroForth, the annual European Forth Conference is celebrating its tenth anniversary this year in England. The Conference title, "Exploiting Forth", reflects the need in todays economic climate to make the best use of all the features that Forth provides. In particular this year's conference will show all the benefits and capabilities of merging Forth with modern programming environments.

Conference delegates are welcome and encouraged to give papers on subjects related to the conference topics. These papers should be no more than 6 pages. Suggestions for any topic not listed will be gladly considered. Papers should take between 20 and 25 minutes to deliver including questions.

Abstracts should be submitted as soon as possible for acceptance by the committee. Refereed papers were required by May 31st. Camera ready copy is required by 10 October 1994, so that delegates can receive the papers at the conference. Late papers may be accepted at the discretion of the committee.

EuroForth'94 is being held in a pleasant hotel situated in the heart of Winchester, a lovely medieval city featuring the well known Winchester Cathedral. The city is only a 50 minute train journey from London to Winchester, with easy access from both Gatwick and Heathrow airports. Many activities can be found less than 25 miles away including:

| | | |
|---|---|---|
| Resident delegate ............. | £290.00 + VAT. | (including conference fee, hotel accommodation, and all meals) |
| Nonresident delegate ........ | £215.00 + VAT. | (includes conference fee, lunch on Friday, Saturday, and Sunday |
| Resident visitor ................ | £140.00 + VAT. | (including shared accommodation and all meals.) |
| Student Rate .................... | £192.00 + VAT. | (To obtain this rate you must have a National Union of Students card. Also accommodation will be shared.) |

VAT - Please note that VAT (sales tax) is charged at 17.5% in addition to the above prices.

**Registration:** For further information please contact:
**The Conference Organizer, EuroForth'94;** **c\o Microprocessor Engineering Limited**
**133 Hill Lane, Southampton SO1 5AF, England**
**Tel: +44 703 631441, Fax: +44 703 339691;** **net: mpe@cix.compulink.co.uk**

---

## 1994 FORML Forth Conference

Conference Theme:

## *Interface Building*

**November 25-27, 1994**

Time allotments for presentations will favor early submittials and/or theme relevance.
Abstracts are needed by September 1, 1994

Mail your submissions to:
**Forth Interest Group**
**Attn: Mike Elola**
**P.O. Box 2154**
**Oakland, CA 94621**