$10 Volume XV, Number 6 March 1994 April

FORTH DIMENSIONS

Optimizing '386 Assembler

A Line Editor & History Function

Parallel Forth: The New Approach

Readability Revisited

Serial Communications

Print ZIP Barcodes

Contents

Features

6 Adventures in Serial Communications — *Al Mitchell*

Serial data communications can be many times less expensive than parallel methods. But with far fewer wires for the data, speed penalties can be significant. This discussion of data formats, protocols, and circumventing DOS overhead by directly controlling UARTs with Forth shows that properly implemented serial communications can be pushed to extremely high speeds or sent over distances up to four kilometers.

10 A Line Editor and History Function — *Charles Curley*

Users of skeletal Forth systems: you *can* retrieve and re-execute that series of commands without re-typing them. And just because you aren't using a fully frilled text editor doesn't mean you have to live entirely without editing tools. Boost your programming productivity right where you spend the most time—at the command-line.

15 Parallel Forth: The New Approach — *Michael Montvelishsky*

From Russia, the author provides an extension that brings parallel programming to Forth—even if, for now, you do have only one processor. With it, you can write code processes that not only communicate with each other, but manage productive offspring.

20 Readability Revisited — *Garth Wilson*

Every well-run ship and airplane has its gear stowed and hatches secured. More than aesthetics, these are issues of practicality and safety. Whether or not you have been accused of producing write-only code, refining your program's physical layout will improve its clarity, reliability, and maintainability, and will demonstrate your care and workmanship.

27 Print ZIP Barcodes — *Walter J. Rottenkolber*

Generating postal barcodes, whether to speed your mail or to save money, is an interesting project in applied Forth. It requires careful design and placement of graphics, and understanding five-bit numbers and postal codes up to eleven digits long. This implementation will make your letters compatible with state-of-the-art scanning and sorting technologies.

33 Optimizing '386 Assembly Code — *David M. Sanders*

This article discusses optimization techniques for machine code generated during compilatoin. '386 assembly language is used to illustrate the techniques, but many of are applicable to other processors, including the 68000 family. Certain techniques can especially reduce the amount of machine code generated by Forth compilers.

Departments

4 Editorial ..... ANS Forth announced

5 Letters ..... Something old, something new; Off-line resources.

31 Advertisers Index

42 Fast Forthward..... Forth: always new, despite its age

Editorial

ANS Forth Announced

The following on-line message from Elizabeth Rather will mark a major benchmark in the evolution and public perception of Forth:

> "ANS Forth has been officially approved by X3, the Information Systems sub-group of ANSI responsible for computer languages, hardware, media, etc., and was forwarded to ANSI early in January, 1994, for a two-week letter ballot by the Board of Standards Review (BSR). BSR approval is expected to be automatic, as there were no negative votes from X3. Publication of the official standard is expected in March, 1994.
>
> "With the exception of minor editorial corrections, the approved standard will be the same as dpANS6 published for a typographical review on a number of electronic bulletin boards in August, 1993.
>
> "X3J14, the X3 Technical Committee that developed the Standard, will meet June 20, 1994, in Rochester, NY. The purpose of this meeting will be to respond to any requests for clarification that have come in, organize a mechanism for dealing with such requests over the next few years, and vote to enter a 'dormant' stage until the Standard is scheduled for review in approximately four years' time. Guests at the meeting are welcome. Contact E. Rather, 1-800-55FORTH if you are interested."

For information on how to submit an official "request for clarification," see the complete text of Ms. Rather's announcement—Category 10, Topic 2, Message 180 on GEnie's Forth RoundTable.

Where From Here?

Whether or not you adopt, or even implement, ANS Forth yourself, it will generate a predictable wave of interest in Forth. Think of it as a carrier signal or as surf—either way, the task now is to use its momentum, great or small, to increase the visibility and viability of Forth in general, and of our individual skills, products, and services in particular. This is a perfect time for some strategic planning.

This is also a good time to give key people a one-year subscription to *Forth Dimensions*. Our authors will expose them to current developments and techniques, showing that Forth is very much alive and well, and proving that it is a practical and efficient problem-solving tool. Use the mail-order form or contact the FIG office to send someone a gift membership.

* * *

This issue introduces new Russian FIG member Michael Montvelishky, author of "Parallel Forth: the New Approach." We first met Michael through Internet e-mail, but the special assistance of Jeff Fox expedited the appearance of his article in these pages. We thank both of them, and welcome further contributions discussing the significance of Parallel Forth, as well as its implementation.

Is it in the nature of Forth to be write-only, or is that just the nature of some programmers? Like handwriting, it needs to communicate to others but falls short more often than we care to admit. Garth Wilson's "Readability Revisited" reminds us that good code is in the eye of the beholder as well as in the guts of the machine. Some of his methods are familiar (though not used consistently enough), and some may seem controversial. But, stepping out of the ruts of personal prejudice and unconscious habits, one sees a principle at work that is hard to argue with.

Let us know what you think about these and the other articles in this issue—your letters and e-mail are more than welcome, they help to shape what future issues will bring.

—Marlin Ouverson
MARLIN.O on GEnie, or
marlin.o@genie.geis.com

Forth Dimensions
Volume XV, Number 6
March 1994 April

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

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

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

The Forth Interest Group

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

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

Letters

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!

Something Old, Something New

Dear Mr. Ouverson,

I read Ellis Cooper's article on "The Visible Virtual Machine" (VVM) in *Forth Dimensions* XV/5 with some surprise. The stack picture he proposes is in both of my fig-Forths, and these date back to over ten years ago. Moreover, most of the features in his GUI version of Forth are already present in Frans van Duinen's PDE. This editor was praised highly in *FD* XI/2.

I was fortunate to obtain a copy of PDE. After some debugging and tuning up, I have been using my version for three years now. As in Cooper's VVM system, PDE has two windows. One is a screen editor, and the other displays messages and does command-line functions. It differs from VVM in that the second window displays only, the data is not captured to a file.

The primary work window is the full-screen editor. The editor resembles WordStar with similar cursor movements, editing functions, and block moves. Most programming functions (debugger, disassembler, view, etc.) are activated by placing the cursor at the beginning of the word, and pressing the appropriate function key. You can also copy screens from one file to another, add or delete screens, tag screens for recall, and so on, all with just a keystroke or two.

An integrated work environment puts the fun into programming.

A step-trace function on uncompiled words visualizes the stack. A single keystroke activates it, and the cursor moves from word to word. No need to retype words in the command line. Compiler words can be jumped over and the stack contents adjusted.

The VVM, in character-based form, already exists as PDE.

If Cooper's article is a plea for Forth to have a more convenient, integrated, and intuitive development environment, then I fully agree with him. The original editor of my Forth, Laxen and Perry's F83, is command-line driven. It's as unpleasant to work with as a line editor, and is a big turnoff to anyone new to Forth. An integrated work environment, whether VVM or PDE, definitely puts the fun into programming, and should be part of any modern Forth.

Yours truly,
Walter J. Rottenkolber
P.O. Box 1705
Mariposa, California 95338

Off-Line Resources

Dear Sir,

I recently received *Forth Dimensions* XV/4, within which were a number of useful telephone numbers for on-line resources. I have tried to make contact with the two United Kingdom numbers, but both seem to be inaccurate. When I called the number given for Max BBS, I made contact with an elderly gentleman who was most disturbed by my call, as he has had many similar calls requesting access to the BBS.

Please, can you re-publish the correct numbers as soon as possible, so that I can make contact with those BBS's if they are still active, and also prevent any further annoyance to the gentleman at the 0905 number.

Best regards and keep up the good work.
B.M. Morris

Thanks, Mike, for setting us straight. Just before receiving your note, I had a conversation with the FIG office in which we agreed not to publish any of the reSource Listings, not even the FIG Chapters information, until we can complete the rather daunting task of updating and reconfirming the validity of all that information.

Our apologies to anyone who has answered their telephone to be greeted by a shrieking tone even more irritating and unwelcome than that of a bill collector or telephone salesperson. —Ed.

Adventures in Serial Communications

Al Mitchell
Loomis, California

Serial vs. Parallel

Data communications comes in at least two flavors, serial and parallel. Deciding which is appropriate for an application is often as much an economic as a technical decision.

Serial communications has a significant advantage in hardware costs over parallel communications: very few wires are required. Information is transmitted one bit at a time rather than a byte at a time. This mandates either a longer time to transmit the data or a higher data rate, but the number of wires is significantly reduced. Because fewer wires are needed, the attendant costs are reduced.

Parallel communications transmits a bit on each of many conductors strobed by a separate clock line. This increases the effective data rate as well as the costs. Each bit will require not only a wire but a driver, a receiver, and two connector pins. Additionally, there are usually a number of qualifier signals, including the clock. Further, due to the number of active lines within the cable, interference is increased, limiting the upper bandwidth more than is immediately apparent. The cost of parallel communications hardware can be many times that of serial communications.

The cost of parallel hardware can be many times that of serial.

Bandwidth Limitations

Let us assume similar impedance characteristics for both serial and parallel cables. The most common parallel standard is the Centronics Printer Interface. In this standard, information is sent in eight-bit bytes. While the data on one line may be rising, the adjoining wire may be falling. There is interference between the two which limits the upper data rate.

If a cable with similar characteristics were used for serial communications, the above problem would be less significant, as only one wire is changing state at a time. In most serial communications schemes, the data is self clocking via the protocol used.

Let's first look at the most common protocols.

Serial Data Formats and Protocols

Synchronous data transmission is usually reserved for machine-to-machine communications, due to the stability required. Most operator interfacing is by keyboard, which gives the data transmission rate a sporadic nature; therefore, we will not cover synchronous communications in this paper.

Most computers are set up for asynchronous serial communications. The ubiquitous IBM computers use a Universal Asynchronous Receiver/Transmitter (UART) to reduce CPU overhead. The UART, usually an 8250 or 16550, is a semi-intelligent peripheral which has a number of programmable functions. Hardware connections and functions are shown in Figure One.

Only a few of these many wires and signals are required to effect reliable communications. The original RS-232 standard was written in 1969, and room was left for future enhancement, which is seldom required. Simple communications in one direction can be accomplished with as little as two wires (Signal Ground and either TXD or RXD), and bi-directional communications with only three wires.

Although the RS-232 hardware is quite simple, a plethora of software options are possible. The 8250 UART can operate from 300 to 115K baud; however, MS-DOS only supports 300 to 9600 baud. This is not a serious limitation since, in Forth, we have the ability to speak to memory-mapped hardware directly. Not only can we use baud rates other than what DOS limits us to, but we have access to a number of input and output lines which can be used for other purposes.

The UART I/O's are normally used, if at all, for handshaking in the expanded protocols of the RS-232 specification, however this is extremely rare. Since the UART I/O's are available, why not use them for hardware expansion? For example, a simple switch could be connected between CTS and RTS to sense the closure of a security door.

All registers of the UART are listed in Figure Two along with their individual bit function. To convert the register address to a hardware address requires an offset described by the COM port designation. For example, let us suppose we want to read the CTS pin on COM1 directly. CTS resides

Figure One. Cabling and pinouts.

	Signal Name	DB25 pin	DB9 pin	Origin
Ground		1		
Transmitted Data	TXD	2	3	DTE
Received Data	RXD	3	2	DCE
Request To Send	RTS	4	7	DTE
Clear to Send	CTS	5	8	DCE
Data Set Ready	DSR	6	6	DCE
Signal Ground	SG	7	5	
Carrier Detect	CD	8	1	DCE
Secondary Carrier Detect		12		DCE
Secondary CTS		13		DTE
Secondary TXD		14		DTE
Transmit timing		15		DCE
Secondary RXD		16		DCE
Receive timing		17		DCE
Secondary RTS		19		DTE
Data Terminal Ready	DTR	20	4	DTE
Signal Quality		21		DCE
Ring Indicator	RI	22	9	DCE
Data rate selector		23		DTE
Data rate selector		24		DCE

in the Modem Status Register, the sixth register, in bit four. We would then add the base address of the respective COM port; in this case, the base address of COM1 is at hex 03FA, so we would add 6. Therefore, to examine the CTS input, in Forth, we could code it as:

```
CONSTANT COM1 $3FA       CONSTANT IIR 2
CONSTANT COM2 $2FA       CONSTANT LCR 3
CONSTANT COM3 $3EA       CONSTANT MCR 4
CONSTANT COM4 $2FA       CONSTANT LSR 5
CONSTANT RBR  0          CONSTANT MSR 6
CONSTANT THR  0          CONSTANT DLL 7
CONSTANT IER  1          CONSTANT DLM 8

: ?CTS              (  -f)
      COM1  MSR    +   P@
      $10  AND    0<> ;
```

Another advantage of using the resident 8250 UART to examine external events are the Delta bits in the Modem Status Register. Often it is necessary to attend to other housekeeping tasks and the program is not able to spend 100% of the time monitoring the individual input. If the input changes state and returns before the program can return then the event will be missed. The Delta bits indicate whether the respective bit has changed state since the last time it was read by the CPU. This often eliminates the need for interrupt-driven hardware.

Serial Data Train

Previously we mentioned that the UART was programmable to different protocols. Which protocol to use is determined by a number of factors. For example, if we were to transfer ASCII data we could use seven data bits, since the standard ASCII character set is confined to 128 characters. Alternatively, to transfer binary data, eight bits are needed to represent the full value possible in an eight-bit byte.

This is why Intel and Motorola designed their Intel Hexadecimal and Motorola-S records. Both are ASCII representations (seven data bits) of hexadecimal code (eight data bits) which can be transmitted over serial lines using seven-bit protocols. Most EPROM burners which operate from a serial port require one of the two formats. ASCII transmission is often used, even though it requires processing of the data, because it is faster to send with one less bit per byte and no error checking.

An RS-232 data byte is composed of one start bit; seven or eight data bits; then an optional parity bit; and one, one and one-half, or two stop bits. The start bit is mandatory, used as a reference for synchronization and preparation of data reception. Depending upon the protocol, each byte can then have between nine and twelve bits.

The parity bit is an interesting subject in itself. Even when the protocol in use requires the parity bit, it is often ignored by the software. This is usually because of the relatively slow interrupt service routines in the 80x86 chipsets.

Inside the clone is an Intel 8259 Programmable Interrupt Controller which receives an interrupt request from the 8250 UART. The 8259 must then notify the CPU that an interrupt has occurred. The CPU then pushes a few registers onto the CPU stack for safekeeping, and calculates the hardware interrupt vector location based upon the interrupt number. The CPU then looks up this 32-bit address and unconditionally jumps to that service address. That address is installed by the communications program being run.

In our case, we will be writing the program so on entry we must know what interrupt number our hardware will be toggling, as well as the hardware address the service routine will reside at. The interrupt service routine will be ended with a ReTurn from Interrupt (RTI) to relinquish control of the CPU and allow normal program flow to continue.

Figure Two. 8250 UART register bit assignments.

Registers		Bits							
		Zero	One	Two	Three	Four	Five	Six	Seven
Receive Buffer	RBR	DB0	DB1	DB2	DB3	DB4	DB5	DB6	DB7
Transmit Holding	THR	DB0	DB1	DB2	DB3	DB4	DB5	DB6	DB7
Interrupt Enable	IER	data ready	TX empty	RX status	modem status	0	0	0	0
Interrupt ID	IIR	0= pending	int ID bit zero	int ID bit one	0	0	0	0	0
Line Control	LCR	Word Length bit zero	Word Length bit one	# of stop bits	Parity enable	Even Parity	Stick Parity	Set Break	Divisor access bit
Modem Control	MCR	DTR	RTS	OUT 1	OUT2	LOOP	0	0	0
Line Status	LSR	DSR	Over-run error	Parity Error	Frame Error	Break Int.	TX Hold Reg.	TX Shift Reg.	0
Modem Status	MSR	Delta CTS	Delta DSR	Trail Edge	Delta DCD	CTS	DSR	RI	Delta CD
Divisor Latch Low	DLL	0	1	2	3	4	5	6	7
Divisor Latch Hi	DLM	8	9	10	11	12	13	14	15

As you can see, the overhead required is substantial. The program should be optimized before attempting any high data rates. Even a '386-class machine is hard pressed to run 9600 baud reliably. In Microsoft Windows we have found a '386\40 machine insufficient to run 9600 without error.

Because of the tremendous overhead of the ISA Bus machines, we need to consider that the CPU will occasionally be busy in other tasks when the UART signals that a byte has been received. That byte needs to be rescued as fast as we possibly can to avoid being lost. If we do not respond to the interrupt request fast enough, the next incoming byte of data will overwrite the first, forever to be lost unless we implement one of the more esoteric error detection and recovery methods.

At 9600 baud, with no parity bit, eight data bits, and one stop bit, this is one byte every

$$\text{bytes-per-second} = \frac{\text{one second}}{[9600/(1\text{ start bit} + 8\text{ data bits} + 1\text{ stop bit})]}$$

or one byte every 1,041 microseconds.

Further overhead is caused by DOS calls for writing to the disk or to the screen, which are general-purpose ROM BIOS routines and not optimized for speed. Faced with these combinations of overhead, we elected to create a temporary ring buffer to receive the incoming data until the foreground task can complete its processing.

A Simple Communications Program

To minimize conflicts with the foreground tasks, let us first create a ring buffer large enough so that we can be away for extended periods of time:

```
1024 constant serial-buffer-length
create serial-buffer
       serial-buffer-length allot
variable head
variable tail
serial-buffer serial-buffer-length +
       constant end-of-serial-buffer
```

This creates a receive buffer into which the interrupt routine will pass the received data. The variables `head` and `tail` bracket the input data for foreground processing.

The interrupt routine is relatively straightforward—we need only to read the UART receive buffer, increment the tail pointer, and write the received byte at the tail pointer address, then return from the interrupt (see Listing One).

The standard COM3&4 DOS mapping shares INT3 with COM2&4, and INT4 with COM1&3. This is very unstable. We use a COM3&4 board which has the ability to map the interrupts to any of the 15 AT-class interrupts, and use INT10&11 for COM3&4, respectively. The CICC232 board from B&B Electronics gives complete freedom as to the interrupt selection, hardware address of the ports, selection of RS-232/422/485, and uses an enhanced UART, the 16550. The 16550 features a 16-byte FIFO to spool the incoming data and is highly recommended, especially at the higher data rates or within Windows. (See Listing Two.)

Next, we compute and install the interrupt vector (Listing Three).

Putting it all together, here is a simple terminal program:

```
:      DUMB          (  -)
              INITS
              BEGIN
                    BYTE?   IF EMIT THEN
                    KEY?    IF TXD THEN
              UNTIL ;
```

RS-232

The hardware description is a single-ended digital signal of at least ±8.0 volts output. The inputs are defined as having thresholds of ±3.0 volts. This gives a very high noise rejection and hysteresis, which results in excellent performance up to about 50 feet at 9600 baud.

RS-422 and RS-485

RS-422 uses the same UART and identical software of the RS-232 standard, while the RS-485 requires one additional signal. The RS-485 defines a "Party Line" or "Multi-Drop" communications method allowing up to 32 talkers/listeners to hang on the same set of lines.

Allowing multiple talkers requires that each node, when not specifically speaking, be in a receive, not transmit, state. This is sometimes called "tri-stated." If two talkers simultaneously attempt to speak on the same line, excessive current will flow and the data from both will be garbled. This requires extra hardware to gate the transmitter on and off while the receiver is turned off and on. In the B&B COM board used, 232CICC1 is used to provide this function.

Both of these standards refer to balanced transmission lines rather than the single-ended (unbalanced) line of the RS-232 standard. Balanced lines are a pair of lines which transmit opposite polarity signals simultaneously to represent the serial data.

Balanced lines have much higher data integrity. The net current through the pair is a null value, so the interference output is reduced as well. Standard drivers typically pull one line to +5 volts while the other is pulled to ground. A bit change will reverse the levels, giving an effective ±10 volt transition from low voltage logic-level power supplies.

Due to the balanced lines, more powerful drivers, and sensitive receivers, distances can be extended to as much as four kilometers under ideal circumstances, or to very high data rates over shorter distances.

Listing One.

```
label serial(com1,2)
      ax push   bx push   dx push
      BEGIN   CS: COM-PORT #) DX MOV
            DX AL IN   4 # AL TEST
      0<> WHILE   CS: COM-PORT #) DX MOV
            2 # DX SUB   dx al in
            cs: head #) bx mov   cs: al 0 [bx] mov
            bx inc      end-of-serial-buffer # bx cmp
            0= if   serial-buffer # bx mov   then
            cs: bx head #) mov
      REPEAT
      $20 # dx mov   $20 # al mov   al dx out
      dx pop   bx pop   ax pop   iret   end-code
```

Listing Two.

```
label serial(com3,4)
      ax push   bx push   dx push
      BEGIN   CS: COM-PORT #) DX MOV
            DX AL IN   4 # AL TEST
      0<> WHILE   CS: COM-PORT #) DX MOV
            2 # DX SUB   dx al in
            cs: head #) bx mov   cs: al 0 [bx] mov
            bx inc   end-of-serial-buffer # bx cmp
            0= if   serial-buffer # bx mov
            then   cs: bx head #) mov
      REPEAT
      $20 # al mov   $A0 # dx mov   al dx out
      $20 # dx mov   al dx out   dx pop
      bx pop   ax pop   iret   end-code
```

Listing Three.

```
CREATE      DOSINTS     $0C C,   $0B C,   $72 C,   $73 C,
: DOSINT#   (  - b)          COM @ 2/ DOSINTS + C@ ;

: SET-SERIAL-VECTOR     (  - )
      ?cs: DOSINT# $72 <
      IF    serial(com1,2)
      ELSE  SERIAL(COM3,4)
      THEN  DOSINT# interrupt! ;

CREATE INT#S
      $EF C, $21 C, $F7 C, $21 C, $FB C, $A1 C, $F7 C, $A1 C,

: SET-8259  (  - n a)
      COM @ INT#S +
      COUNT >R C@   DUP PC@ R>   AND SWAP ;
```

Al Mitchell has been doing embedded systems programming since 1974, when he worked on the Intel 4004.

A Line Editor and History Function

Charles Curley
Gillette, Wyoming

This paper describes an input line editor, á la MS-DOS, and a history function, á la Unix, giving Forth the best of both.

Historical Note

The Forth used for the code described herein is FastForth, a full 32-bit BSR/JSR-threaded Forth for the 68000, described in unmitigated detail in *Forth Dimensions* XIV/5. It is a direct modification of an indirect-threaded Forth, real-Forth. This is, in turn, a direct descendent of fig-Forth. (Remember fig-Forth?) fig-Forth vocabulary, word names, and other features have been retained.

For those not familiar with 32-bit Forths, the memory operators with the prefix `W` operate on word, or 16-bit, memory locations. FastForth uses the operators `F@` and `F!` for 32-bit memory operations where the address is known to be an even address. To avoid odd address faults, the regular Forth operators @ and ! use byte operations.

The FastForth version for the 68000 is shown here, rather than an MS-DOS 80x86 version, to avoid the confusing and ugly overhead of segmented architecture.

Often, an error in a command is revealed only after the user hits return.

Why Bother?

The basic intent behind this exercise is to give Forth two facilities the implementor has found useful in other operating systems. A line editor, such as that provided by MS-DOS, allows the user to modify a command line by moving the cursor, and inserting and deleting characters. Many Unix implementations provide a history function, which allows the user to recover previously executed command lines and re-execute them. This and the line editor capability combine to give FastForth very powerful command processing.

Charles Curley is a long-time Forth nuclear guru who lives in Wyoming. When not working on computers he teaches firearms safety and personal self defense. His forthcoming book, *Polite Society*, covers federal and state firearms legislation in layman's terms.

Using the Line Editor and History

The line editor can be very useful. Often, as a Forthwright enters a line of text, he finds he has committed a typo. With the normal `EXPECT`, one would have to backspace to the error, correct it, then re-type that portion of the line which was backspaced out. The traditional `EXPECT` did not erase characters on the screen as the backspace key was entered, giving the neoForthwright the impression that what he was backspacing over had been retained, when in fact it had been discarded.

This line editor eliminates that false impression, and also makes it possible to retain a portion of the line to the right of the cursor while making corrections.

For example, if you want to dump a portion of memory, you will need to enter a starting address, a count, and the word `DUMP`. If you are just about to hit return, and realize that you want to force the address and count to be in hexadecimal, you can left-arrow to the beginning of the line, enter the word `HEX`, a space, and hit return. The complete line will be there, and you will get your hex dump.

Often, an error in a command line is revealed only after the user hits return and the system executes the command line. In the above example, you can use the history function to recall the dump command. You can then edit the line to dump at a different address, or for a different count, or in a different base.

This is useful when you forget which vocabulary you are in. If you enter a command and a word is not currently available, the command will fail. You may then recover the command, prepend a vocabulary name to it, and hit return.

The history function also allows the user to define and edit macros on the fly. These macros are previous entries in the string array. Repetitive searches of databases or through source code can be implemented by typing the first command, and repeating it with the up-arrow key.

The Design

FastForth is intended for embedded processor applications. A minimal version of `EXPECT` is suitable for such a nucleus. In such a version, only the backspace and delete keys operate, and they operate identically.

However, FastForth's EXPECT is vectored, so programmer utilities such as the assembler and editor may include a more extensive line editor capability. This allows extensive line editing on the target hardware during development without sacrificing nucleus size. The line editor is removed before the program is committed to ROM.

The line editor uses a number of variables to track the size of the line being edited, the location of the cursor, and other characteristics. These are annotated in detail in the source code and in the glossary.

A headerless string array is defined for the history capability. This array stores each string in the history in the traditional Forth count-and-string format. The starting address of each entry is stored in another array, STRARRAY. This is an array of pointers. A variable, NXTLN, points to the current string indirectly by pointing to the string array. This means that moving from each entry to the next consists of incrementing or decrementing NXTLN by the size of a pointer, and handling wrap-around correctly.

The traditional Forth array was not used here because of speed considerations. This would have been defined something like this:

```
: ARRAY CREATE STRINGS * ALLOT
    \ count ---   | build array
  DOES> >R  STRINGS *  R> + ;
    \ entry# --- addr |
```

This means that every access to the array would have involved a multiply. The double-indirection method eliminates this time-consuming operation. Furthermore, using the double-indirection method means that calculations are performed only at the time a new line is selected, never at access time.

Using these variables and indirection, strings are loaded into the string array and copied from it to the edit buffer by always referring to the variable.

A design decision was made that, when a line is copied from the history buffer, the cursor will appear on the left end of the line. This is due to experience with Forth. Often a line must be edited and re-executed because a vocabulary or other modifier was not given. These modifiers typically must be at the beginning of the line, not the right-hand end. Also, the Home key provides a quick method for the user to move to the right-hand end of the line, should he wish to.

The Implementation

The glossary indicates the function of each word. The listing should be read along with the coding descriptions of the words given below.

The implementation starts with a loader screen, screen 501. For development purposes, the debugger can be called. For inclusion in the utilities, this call to the debugger is commented out.

The word TASK is used as a marker in the dictionary. During development, this is forgotten and recompiled below the application to allow the programmer to recompile the application by entering the phrase RETRY TASK. Once an application is deemed complete and is to be added to the utilities, TASK is forgotten, the application is compiled, then TASK is redefined on top. Used in this manner, the word acts as a moveable marker to indicate where the utilities end and applications begin.

The word BELL defines a word which emits an ASCII bell character. This is typically used as an alarm or an alert on some error.

Line five of screen 501 forces the line input code to be (EXPECT), the nucleus input working word. This is done at compile time to ensure (especially during debugging) that, in the event of a compile-time error, EXPECT will have a valid working word to execute.

On line seven, we establish the vocabulary where most of the line editor will reside. This hides the bones of the application from the user once the project is completed.

Line nine of screen 501 effects compilation of the next six screens. It is written this way because the word +BLK and its family may not exist when this code is compiled into the utilities.

Screen 502 compiles a number of variables and constants. These are described in the glossary. The most important one to the user is the constant STRINGS, on line seven. It indicates the number of strings to be preserved in the string array. It is the equivalent of the Unix environment variable HISTORY. In order to effect changes in this constant, the line editor must be recompiled.

On line 11, a most implementation-specific constant sets the maximum line length. This constant should specify the maximum line length that the user may enter into the terminal input buffer in the course of normal Forth operations.

The next screen, 503, shows the construction of the string array by the word MAKESTRS. This word allots memory for each string in the buffer, and stores its address in the array of string addresses, STRARRAY. This word is executed once, at compile time.

The next two words handle the process of moving from one entry in the string array to the next. UP moves from a given string to the next higher address string, and handles wrap-around at the top. DN defines the process for moving down in the string buffer. These two words turn the string array into a ring buffer for strings.

The phrase NXTLN F@ F@ will show up often in the remainder of the code, as that phrase always returns the address of the current string buffer entry. In a traditional Forth, this phrase would probably be built into a word to make the code more readable. However, the FastForth optimizing compiler turns this three-word phrase into two processor instructions. The implementor chose to go for faster code over programming elegance.

Similar phrases based on variables, such as CURS F@ and SIZE F@, resolve to one processor instruction, and so are not built into words that make pseudo-constants.

HISTORY, and its alias HIST, at the top of screen 504, show the ring string buffer at work. They exist to allow the user to view the contents of the string array. They are made generally available to the user by placing them in the FORTH vocabulary. HISTORY walks through the ring buffer, dis-

Glossary

Name	Screen	Line	Vocabulary	Flags

BELL 501 3 FORTH u
Emits the standard ASCII bell character to the current output device.

BKSP 506 1 EXPECTING u
Deletes the character to the left of the cursor. If the cursor is at the left edge, a bell is emitted and no other action is taken. It is activated by the Backspace key.

BUF 502 2 EXPECTING i u
This variable holds the address of the buffer in which text is being edited.

CURS 502 1 EXPECTING i u
This variable holds the relative position of the cursor within the line of text being entered.

DEL 505 10 EXPECTING u
Deletes the character under the cursor. It emits a bell if the result is an empty line. It is activated by the Delete key.

DN 503 9 EXPECTING u
Moves down one line in the string buffer, and places that line in the edit buffer for editing. It is activated by the down-arrow key.

EXPECTING 501 7 FORTH i u
This vocabulary hides the bones of the line editor from the user.

GETSTR 505 1 EXPECTING u
Recovers a string from the string array and places it in the editing buffer. This code replaces the existing contents of the editing buffer, so the cursor is forced to the left edge of the terminal.

HIST 504 4 FORTH u
An alias for HISTORY. Some versions of Unix use this name.

HISTORY 504 1 FORTH u
A utility word to print out the contents, if any, of the string buffer. The name is taken from the Unix utility with much the same function.

INSERTKEY 506 13 EXPECTING u
This function toggles the insert flag. It is activated by the Insert key.

INSRT 506 7 EXPECTING u
Insert a given key into the edit buffer at the current cursor position. To overwrite, use PUT.

LEFT 505 5 EXPECTING u
This function moves the cursor left one position, non-destructively. If the cursor is already at the left side of the buffer, a bell is emitted instead. It is activated by the left-arrow key.

LNED 507 1 FORTH u
This is the application word. It is substituted for the working word of EXPECT in order to activate the history and line editing functions.

Name	Screen	Line	Vocabulary	Flags

MAKESTRS 503 1 EXPECTING u
This word runs once at compile time. It builds the string array and the array of pointers to the strings.

NXTLN 502 5 EXPECTING i u
This variable points to the next location in the string buffer to be used.

PLACE 506 11 EXPECTING u
Put a character into the current line being edited, in either overwriting or inserting mode.

PLACEF 502 4 EXPECTING i u
This flag indicates whether the line editor is in overwriting mode (zero) or in inserting mode (non-zero).

PUT 506 3 EXPECTING u
Place a character into the current string in overwriting mode.

REDRAW 504 7 EXPECTING u
Redraws the line on the screen. It is typically used after inserting or deleting a character.

REND 505 14 EXPECTING u
This function moves the cursor to the right end of the line being edited. It is activated by the Home key.

RIGHT 505 7 EXPECTING u
This function moves the cursor right one place in the current line. If the cursor is already at the right end, a bell is emitted instead. It is activated by the right-arrow key.

SIZE 502 3 EXPECTING i u
This variable holds the current size of the line already in the edit buffer. It is adjusted as the string is expanded or contracted.

STORESTR 504 13 EXPECTING u
This function places the line in the edit buffer into the string array. It is activated when editing is complete.

STRARRAY 502 9 EXPECTING i u
This array holds pointers into the string array.

STRINGS 502 7 EXPECTING i u
This constant indicates the number of strings to be held in the string array. It is used at compile time and run time.

STRSIZE 502 11 EXPECTING i u
This constant indicates the maximum size of the strings to be edited. It is derived in a very system-specific manner from the nucleus word QUERY.

UP 503 6 EXPECTING u
This word moves up one line in the string array. It is activated by the up-arrow key.

(This glossary was partially produced by a glossary generator which is part of FastForth.)

```
Scr #   501
 0 \ begin line editor     ( 16  5 91 CRC  8:25 )
 1 FORTH DEFINITIONS   (   DEBUG      )    FORGET TASK
 2 FORTH DEFINITIONS   (   : TASK ;   )    BASE F@ DECIMAL
 3 SEQ BELL  1 C,  CTL G C,
 4
 5 ' (EXPECT) 'EXPECT F!
 6
 7 VOCABULARY EXPECTING IMMEDIATE  EXPECTING DEFINITIONS
 8
 9   BLK F@ DUP 1+ SWAP 6 + THRU   \ HERE FIRST OVER - ERASE
10
11   ' LNED 'EXPECT F!
12   HERE FENCE F!   BASE F!       : TASK ;        EDITOR FLUSH
13
14
15

Scr #   502
 0 \ line editor: variables, constants  ( 25 11 91 CRC 13:44 )
 1 0 VARIABLE CURS                  \ line cursor
 2 0 VARIABLE BUF                   \ holding buffer
 3 0 VARIABLE SIZE                  \ size of line already in buffer
 4 1 VARIABLE PLACEF                \ are we inserting or replacing?
 5 0 VARIABLE NXTLN                 \ pointer to next line to place
 6
 7 10 CONSTANT STRINGS              \ number of back strings saved
 8
 9 CREATE STRARRAY STRINGS 4* ALLOT        \ pointer array
10
11 ' QUERY 5 + C@ CONSTANT STRSIZE \ *very* implementation specific
12
13
14
15

Scr #   503
 0 \ line editor: makstrs, up dn history     ( 25 11 91 CRC 13:47 )
 1 : MAKESTRS      \ construct the string buffer at compile time.
 2   STRINGS 0  DO  HERE  DUP  STRARRAY I 4* +  F!
 3       STRSIZE 1+  DUP ALLOT  ERASE  LOOP ;
 4 MAKESTRS        STRARRAY NXTLN F!
 5
 6 : UP    4 NXTLN +!  NXTLN F@ STRARRAY - 4/
 7   STRINGS = IF STRARRAY  NXTLN F! THEN ;
 8
 9 : DN    NXTLN F@ STRARRAY    \ down one in the string buffer
10   = IF [ STRARRAY STRINGS 4* + ] LITERAL NXTLN F! THEN
11   -4 NXTLN +! ;
12
13
14
15

Scr #   504
 0 \ line editor: redraw, history access     ( 25 11 91 CRC 13:47 )
 1 FORTH DEFINITIONS
 2 : HISTORY  EXPECTING STRINGS 0 DO  DN  NXTLN F@ F@
 3   COUNT DUP IF CR  I 4 .R SPACE THEN  TYPE  LOOP  SPACE ;
 4 : HIST HISTORY ;
 5
 6 EXPECTING DEFINITIONS
 7 : REDRAW \   ---   | re-display the line from cursor
 8   ERL  BUF F@  CURS F@ +  SIZE F@  CURS F@ -
 9   -DUP IF  DUP >R  TYPE   SPACE
10        R> 1+  0  DO  LEFT  LOOP
11        ELSE  SPACE LEFT  DROP  THEN ;
12
13 : STORESTR      \ --- | store string in array
14   BUF F@  NXTLN F@ F@ 1+  SIZE F@  2DUP SWAP 1- C!  CMOVE ;
15
```

playing each line in it. Since the word walks through the ring buffer exactly the number of times that there are entries, the pointer is left back in its starting position. This code depends on the word TYPE dropping its two arguments when presented with a count of zero.

REDRAW uses the terminal-specific operator LEFT to move the cursor left one place on the screen. The Atari ST screen emulates the VT-52 terminal, and the present cursor position cannot be read by the application. Other implementations may be possible where the terminal code permits reading the cursor position. REDRAW redisplays the line from the current cursor position out to the right end of the line. An extra space is emitted in case the most recent keystroke deleted a character in the line.

STORESTR (bottom of screen 504) and GETSTR move strings between the edit buffer and the string array. STORESTR stores the count (from SIZE) in the first byte of the array entry, and GETSTR uses the count to determine the length of the string to be moved into the edit buffer.

LEFT and RIGHT, on screen 505, move the cursor one space to the left or right, if possible. If the cursor is already at the end of the line, a warning bell is emitted instead. These two words use their eponymous cursor control words to move the screen cursor appropriately. They are activated by the appropriate cursor-control arrow key.

DEL, on line ten, deletes the character under the current cursor location. Line 11 does the actual deletion, culminating in the CMOVE at the end. Line 12

determines the appropriate reaction. If the cursor is at the left edge of the line, a bell is issued. Otherwise, the size of the string is reduced by one. The line is then redrawn on the screen. It is activated by the Delete key.

At the bottom of screen 505, REND doesn't tear anything. Rather, it moves the cursor to the right end of the string under edit. This function is activated by the Home key.

At the top of screen 506 is the word BKSP which operates the backspace key function. It moves the internal cursor to the left one space, then deletes the character under it.

PUT, INSRT, and PLACE control placement of characters into the edit buffer, according to the flag variable PLACEF. If PLACEF is asserted, the given character is inserted into the string, using INSRT. Otherwise, it overwrites the character under the cursor with PUT. Those two words handle the logical cursor and string size as appropriate.

Control of the placement flag is through the word INSERTKEY. This word toggles the variable flag PLACEF using the word TOGGLE, a byte operator. Because the 68000 places the least significant byte of a long word at the high end of memory, it is the last byte in the cell which must be addressed. This means that the actual address to be toggled is not PLACEF but PLACEF 3 +. The addition is done at compile time here. Using this technique instead of the variable itself costs nothing at run time, because FastForth variables are compiler directives which compile literals into the dictionary.

A more transportable way to code this would be:

```
: INSERTKEY
  PLACEF  DUP F@
    IF   OFF
    ELSE  ON
    THEN ;
```

(Text continues on page 17.)

```
Scr #   505
 0 \ line editor: keystrokes      ( 16  5 91 CRC  8:21 )
 1 : GETSTR \ --- | get string from array
 2   NXTLN F@ F@  COUNT SIZE F!  BUF F@  SIZE F@  CMOVE
 3   CURS OFF  CTL M EMIT  REDRAW ;
 4
 5 : LEFT          \   ---   | action on left arrow
 6   CURS F@  IF  LEFT  -1 CURS +!  ELSE  BELL  THEN ;
 7 : RIGHT  \   ---   | action on right arrow
 8   CURS F@  SIZE F@ -  0< IF  CURS 1+!  RIGHT
 9         ELSE  BELL  THEN ;
10 : DEL          \   ---   | delete under cursor
11   BUF F@  CURS F@ +  DUP 1+  SWAP  SIZE F@  CURS F@ -  CMOVE
12   CURS F@  SIZE F@ -  0< IF  -1 SIZE +!
13       ELSE  BELL  THEN  REDRAW ;
14 : REND         \   ---   | move cursor to right end
15   SIZE F@  CURS F@ - -DUP IF  0 DO  RIGHT  LOOP  THEN ;

Scr #   506
 0 \ line editor: keystrokes      ( 18  1 91 CRC 21:14 )
 1 : BKSP    LEFT  DEL ;
 2
 3 : PUT           \ c ---   | put the char in the buffer
 4   DUP EMIT  BUF F@ CURS F@  + C!   SIZE F@  CURS F@  =
 5   IF  SIZE 1+!  THEN  CURS 1+! ;
 6
 7 : INSRT  \ c ---   | insert the char in the buffer
 8   BUF F@  CURS F@ +  DUP 1+  SIZE F@  CURS F@ -  <CMOVE
 9   SIZE 1+!  PUT  REDRAW ;
10
11 : PLACE   PLACEF F@ IF  INSRT  ELSE  PUT  THEN ;
12
13 : INSERTKEY   [ PLACEF 3 + ] LITERAL 1 TOGGLE   BELL ;
14 FORTH DEFINITIONS
15

Scr #   507
 0 \ line editor: line editor    (  3 12 91 CRC 22:06 )
 1 : LNED          \ addr ct  --   | expect w/ line editing
 2   EXPECTING  >R  BUF F!  CURS OFF  SIZE OFF   NXTLN F@
 3   BEGIN   KEY   DUP 0= IF   KEY  CASE
 4         ASCII R OF      INSERTKEY       ENDOF   \ insert key
 5         ASCII P OF      DN GETSTR       ENDOF   \ dn arrow
 6         ASCII M OF      RIGHT           ENDOF   \ r. arrow
 7         ASCII K OF      LEFT            ENDOF   \ l. arrow
 8         ASCII H OF      GETSTR UP       ENDOF   \ up arrow
 9         ASCII G OF      REND            ENDOF   \ home key
10         ENDCASE   ELSE  DUP CTL M -
11     IF  DUP 127 = IF  DEL  ELSE  DUP CTL H = IF  BKSP  ELSE
12             DUP PLACE  THEN  THEN  THEN  THEN  ?STACK
13     0  BUF F@ SIZE F@ + C!  CTL M =   SIZE F@  R =  OR UNTIL
14   RDROP   REND  SIZE F@  OUT F!  SPACE
15   NXTLN F!  SIZE F@  IF  DN STORESTR  THEN ;

Scr #   508
 0 \ line editor: test setup      ( 23 12 90 CRC 12:28 )
 1 \ : tst pad 30 lned history ;
 2
 3 : TEST  EXPECTING  HEX
 4   PAD 40 ASCII Z FILL  PAD 20 LNED  PAD 40 DUMP SIZE F@ . ;
 5
 6 PAD  BUF F!  10 SIZE F!
 7 PAD  80 ASCII Z FILL
 8 0 +BLK BLOCK  PAD C/L CMOVE
 9
10 ;s
11 : GETSTR \ --- | get string from array
12   NXTLN F@ F@  COUNT  DUP >R  BUF F@  SIZE F@ +  SWAP  CMOVE
13   R> SIZE +!  REDRAW ;
14
15
```

Parallel Forth: The New Approach

Michael Montvelishsky
Saransk, Russia

As a rule, the arrival of a new programming language is caused by the arrival of some new programming method or paradigm. Thus, Algol-60 marked the appearance of the structured programming approach, and Pascal heralded the advanced user-defined data types. Object-oriented programming was born coupled with Smalltalk, but C++ has marked its entrance into the professional leagues. New programming paradigms demand new programming languages. But Forth users need not change languages because, of course, Forth is extensible and can easily adopt new paradigms. The subject of this paper demonstrates how Forth can adapt a parallel programming paradigm.

One of the hot areas of programming science is "parallel computing." C.A.R. Hoare, author of *Communication Sequential Processes* is the first theoretical founder of parallel computing. The programming language Occam is based on Hoare's theory. It is considered to be one of the main parallel-programming languages. There are two major features of this interesting language:

Forth users need not change languages—because Forth is extensible, it can easily adopt new programming paradigms.

1. Channels for information exchange and synchronization between processes.
2. The ability of the currently running process to run several (the number may be very large) "son" parallel processes. Parent processes stop themselves and run son processes, then they start to run again after all son processes are done.

Another popular parallel programming model is Linda by D. Gelernter. Linda is not a programming language, but a parallelizing extension to a conventional programming language. There is C-Linda, Fortran-Linda and Forth-Linda. Linda's paradigm is based on the idea of active and passive tuples. Active tuples are processes started by some other process and running with it cooperatively. Passive tuples are tools for information exchange; they exchange information like notes on a bulletin board.

Paradigms of Occam and Linda do not conflict with one another, but rather complement each other. It is hard to emulate Linda in Occam, and it is hard to emulate Occam in Linda efficiently.

The subject of this paper is a parallel extension to Forth. What new has arrived with Parallel Forth? Very little: see the glossary on the next page.

Several notes, about the source code:

1. All the user variables of a son that is defined in a program have the same values as those of its parent before `| | ( ... )` or `EV ( ... )` constructions. Thus, it's possible to use user variables to pass parameters from a parent to a son, or to a cooperative parent.
2. Make sure the user area and stack size for each process are big enough to run without hanging up. It's easy to write special words to fill these areas with some character and then to see how many of them were used.
3. If you mount a son process' ring with `| | ( ... )`, you can't run co-operatively with `EV ( ... )` until `PAR` starts the ring where the `EV ( ... )` is at.
4. There is automatic memory allocation for `| | ( ... )` and `EV ( ... )` processes. But automatic memory deallocation is only for `| | ( ... )` processes (and all processes inside them)! Can you guess why?
5. All processes start with empty stacks.

Now, about the programs. The source of the parallel Forth extension is in Listing One. I've used F-PC ver. 3.53. It's one of most complete Forths for IBM-PC compatibles, and it's public domain! The source doesn't contain code words, so you can transfer it easily to another Forth platform with only minor changes. You can rewrite `PAUSE` in assembler to increase efficiency, but the best way is to implement Parallel Forth on some real multi-processor hardware.

A sample program is shown in Listing Two *[page 19]*. I've chosen a scalar vector multiplication as an example of using the parallel wordset. The task is like task X4 from

Glossary

MULTI ..... Enable multi-processing mode.

SINGLE ..... Disable multi-processing mode.

|| (...) ..... Coupled words to add a new parallel branch (all words inside parenthesis) into the currently mounted (not running!) process' ring.

EV(...) ..... Coupled words to add a new parallel branch into the currently running process' ring. It's like the active tuple in Linda, and the syntax has come from Forth-Linda by Jeff Fox.

PAR ..... Stop the current process and start a son process' ring, mounted by the parent process by means of a || (...) construction. Wait until all son processes are done, then run the parent process again.

PAUSE ..... Deferred (for now!) word. In MULTI mode, switch to the next process in the current ring. In SINGLE mode, this is a no-op.

STOP ..... Stop a process and remove it from the current process' ring.

STOP-ALL ..... Stop all processes in the current ring and resume the parent process. Very useful to emulate Occam ALT processes.

DATA-STACK-SIZE
RETURN-STACK-SIZE
USER-AREA-SIZE ..... Variables to control the size of USER areas and stacks. Needed because different processes have different demands for stacks and the number of user variables.

BUFFER ..... Generic word to define buffers. Use buffers for the most efficient data exchange between processes (without synchronization).

>B (n b --) ..... Output a 16-bit value to the buffer b.

B> (b -- n) ..... Input a 16-bit value from the buffer b.

CHAN ..... Generic word to define channels. A channel is a fixed-length buffer, useful for the exchange of information with synchronization.

[]CHAN ..... Generic word to define channels' area.

>C (n c --) ..... Output a 16-bit value to the channel c.

C> (c -- n) ..... Input a 16-bit value from the channel c.

[]WORD ..... Just like F-PC's ARRAY and with the same aim.

chapter 4.3 of C.A.R. Hoare's *Communication Sequential Processes*. I've included the equivalent Occam program code as comments. Sorry, I have no transputer, so I have not checked out this Occam code but I hope it's valid.

The second example program (Listing Three) is less multi-processing but more useful. It's an alarm clock. Type:

```
21 00 ALARM" It's time to go home!"
```

to start experimenting with Parallel Forth!

Michael Montvelishky is from Saransk (capitol of the republic of Mordovia, 600 km. southeast of Moscow). He received his Electronics Engineer diploma in 1982 at Mordovia State University. He went to Leningrad (St. Petersburg now) University in 1982 for postgraduate work. There he dealt with robotics software and got his Ph.D. degree in 1989. He started with Forth in 1988. "It's my favorite so far, but I write programs in C and Pascal sometimes. There are many projects I've done (alone or in tandem with Max) in Forth. Latest and greatest is the embedded system for CNC of Electroerosion machine tool with High-Level Geometric (Forth-based) language and CAD features. We've stopped work on this (almost finished!) project now, because the customer is on the edge of bankruptcy and hasn't money to pay us. I'm a reader (docent) at Mordovia University now, using Forth in my course of programming."

Jeff Fox adds "Dr. Montvelishsky is currently doing work for Ultra Technology. He has submitted several routines for the MuP21 and F21 microprocessors under development at Computer Cowboys by Charles Moore. Dr. Montvelishsky's CORDIC function executes 50 times faster on MuP21 than on a '387. Here at Ultra Technology, Dr. Montvelishsky will be consulting on the design of parallel Forth extensions for the F21 Parallel Forth engine, robotics, scientific programming, and on using Forth to teach computer science."

Readers can write to Dr. Michael B. Montvelishsky, Avenue Lenin 21-29, Russia 430000, Saransk or send e-mail to mic@fclub.mordovia.su on the Internet.

Listing One. Parallel Forth extension.

```
\ Dr. Michael B. Montvelishsky, Saransk Russia 1993

\ PARALLELISM WORDSET        by Michael Montvelishsky, 25-Sep-93
 ANEW PARALLEL
 CR .FREE
\ ========================================================== \
\ It is just a simple troock      :-) "TROOCK" is  the Russian
\ To replace the pause hook       :-) pronounce of the "TRICK"
\ PAUSE is CODE-word now.
' PAUSE                 \ OLD
DEFER PAUSE ' NOOP IS PAUSE
' PAUSE                 \ OLD NEW
2DUP SWAP -             \ OLD NEW SHIFT
OVER 1+ +!              \ OLD NEW ( ADJUSTED!)
TUCK                    \ NEW OLD NEW
HERE SWAP -             \ NEW OLD SIZE
CMOVE FORGET PAUSE
\ PAUSE is DEFERed NOW!
\ ========================================================== \

USER
VARIABLE (NEST)                 \ Point to the last mounted proc
VARIABLE (PARENT)               \ Point to the parent proc
VARIABLE (NEXT)                 \ Point to the next proc in ring
VARIABLE RP                     \ To save
VARIABLE SP                     \         stacks' value FORTH
 VARIABLE RETURN-STACK-SIZE      \ To change return stack size
 VARIABLE DATA-STACK-SIZE        \    -\\-   data stack size
 VARIABLE USER-AREA-SIZE         \    -\\-   user area size

\ GLOBAL memory allocation tool:
DP CONSTANT (HEAP)
: MALLOC (HEAP) +! ;
: HEAP (HEAP) @ ;

\ Initiate variables:
UP @ (PARENT) !
UP @ (NEXT) !
0    (NEST) !
64   USER-AREA-SIZE !
128  RETURN-STACK-SIZE !
64   DATA-STACK-SIZE !

: PROC-SIZE  ( -- current-proc-size )
  USER-AREA-SIZE @ RETURN-STACK-SIZE @ + DATA-STACK-SIZE @ +
;
: GO-NEST   SP@ RP@ RP ! SP ! (NEST) @ UP ! RP @ RP! SP @ SP! ;
: (PAUSE)   SP@ RP@ RP ! SP ! (NEXT) @ UP ! RP @ RP! SP @ SP! ;

\ Stop all current processes and go to parent process
: STOP-ALL  (PARENT) @ (NEXT) ! (PAUSE) ;

\ Parallelism enable/disable
: MULTI     ['] (PAUSE) IS PAUSE ;
: SINGLE    ['] NOOP    IS PAUSE ;

\ Stop itself
: STOP ( --- : Excludes proc from procs' ring )
  UP @ DUP (NEXT) @ - IF  \ Last in the ring?
    UP @ BEGIN (NEXT) 2DUP @ - WHILE @ UP ! REPEAT
    SWAP UP ! (NEXT) @ SWAP ! (PAUSE)
  ELSE
    STOP-ALL
  THEN
;

\ Create NEW process with specified PFA PARRENT -process and
\ process to be PREVious in processes' ring
```

(Listing One continues on next page.)

(Line Editor, continued from page 14.)

INSERTKEY is activated by the Insert key.

The last screen is 507, and it is wholly given over to the monster word LNED. Line two contains the necessary initialization. The main loop, starting on line three, concludes at the end of line 13. This is followed by the close-out code on lines 14 and 15.

Most of the interior of the loop is a case statement using the Eaker case statement. Lines ten, 11 and 12 deal with the delete and backspace keys in a series of nested `if ... else ... then` statements. Line 13 deals with the end-loop condition, which is either a carriage return (control-M) or an overflow of the line buffer.

Screen 508 contains a number of test words which were found useful during the coding phase. They are not part of the final application, and are not compiled except for testing.

Completion of compilation returns control to screen 501. Line 11 installs the line editor into the system by ticking the line editor and placing its code field address into the user variable 'EXPECT.

Because forgetting this code would have disastrous results, we inhibit forgetting below LNED by moving FENCE to above it.

User Controls

Because the code is self-installing at compile time, the user does not even need to know it is there in order to use it. However, he will get the most out of it by adding a few keystrokes to his repertoire.

The left- and right-arrows operate to move the cursor left and right within the line.

The up- and down-arrows operate to move the user within the string buffer. The up-arrow will retrieve the most recent line from the string buffer into the editor buffer. The user may scroll through the string buffer by repeatedly striking the up-arrow.

```
: MOUNT ( PFA PARRENT PREV --- : Create and mount new process)
  HEAP PROC-SIZE MALLOC                      \ Allocate proc space
  UP @ DUP >R OVER USER-AREA-SIZE @ CMOVE \ Copy parent's users
  OVER UP ! (NEXT) @ SWAP UP ! (NEXT) !    \ Adjusts new  (NEXT)
  UP @ USER-AREA-SIZE @ +
  RETURN-STACK-SIZE @ +                      \ Adjust for the NEW :
  DUP DUP RP0 ! 4 - RP !                     \    RP0 & RP
  DATA-STACK-SIZE @ + DUP SP0 ! SP !         \    SP0 & SP
  SWAP (PARENT) !                            \    (PARENT)
  (NEST) 0!                                  \    (NEST)
  -ROT RP @  2!                              \ PFA to begin with
  UP @ SWAP UP ! (NEXT) !                    \ Point PREV's (NEXT)
  R> UP !                                    \   to the NEW
;

\ Run-time words for mounting and evaluating new process
: (||()
  2R@ 4 + UP @ (NEST) @ ?DUP 0=
  IF HEAP DUP DP ! THEN   HEAP (NEST) ! MOUNT
;

: (EV()
  (NEST) @ 0= IF
    2R@ 4 + (PARENT) @ (NEXT) @ MOUNT
  THEN ;

: ||(  COMPILE (||() COMPILE BRANCH ?>MARK ; IMMEDIATE
: EV(  COMPILE (EV() COMPILE BRANCH ?>MARK ; IMMEDIATE
: )    COMPILE STOP ?>RESOLVE ; IMMEDIATE
: PAR  (NEST) @ IF GO-NEST DP @ (HEAP) ! (NEST) 0! THEN ;

\ Use chans for information exchanging & SYNCHRONISATION
: CHAN 2VARIABLE ;
: C> ( CHAN -- W : Get word from the channel )
  >R BEGIN R@ @ 0= WHILE (PAUSE) REPEAT R@ 2+ @ R>  0!
;
: >C ( W CHAN -- : put word to the channel )
  >R BEGIN R@ @    WHILE (PAUSE) REPEAT R@ 2+ ! R> -1! ;

\ Use buffers for EFFICIENT information exchanging
\ BUFF-SIZE and MASK may be increased!
16 CONSTANT BUFF-SIZE
15 CONSTANT MASK
: BUFF  CREATE 0 , BUFF-SIZE ALLOT  ;
: .OUT 1+ ;          ( IN +0 )
: .BUF 2+ ;
: B> ( BUFF -- W : Get word from the buffer )
  >R BEGIN R@ C@ R@ .OUT C@ = WHILE (PAUSE) REPEAT
  R@ .OUT C@ DUP R@ .BUF + @ SWAP 2+ MASK AND R> .OUT C!
;
: >B ( W BUFF -- : Put word to the buffer )
  >R R@ C@ 2+ MASK AND
  BEGIN DUP R@ .OUT C@ = WHILE (PAUSE) REPEAT
  SWAP R@ C@ R@ .BUF + ! R> C!
;
: 4* 2* 2* ;

\ Generic words for word and channel array
: []CHAN
  CREATE 4* HERE OVER ERASE ALLOT ?STACK
  DOES> SWAP 4* + ;

\ IMHO []WORD is more useful than F-PC's ARRAY
: []WORD
  CREATE 2* HERE OVER ERASE ALLOT ?STACK
  DOES> SWAP 2* + ;

MULTI
CR .FREE
```

The down-arrow scrolls through the buffer in reverse order.

The Insert key toggles the insert mode, from overstriking to inserting.

Delete and Backspace operate to delete characters. The Delete key deletes the character under the cursor, and shortens the line by one for each keystroke. The Backspace key deletes the character to the left of the cursor, moves the cursor left one place, and shortens the line by one character.

Further Expansion

Additions will be left as an exercise for the student. The Atari's Help key could be tied into a help screen for the line editor. The Undo key could be used to obliterate the current contents of the edit buffer and allow the user to start over.

The modifier keys, Control and Alternate, could be used to extend the operation of the keyboard. For example, control-left-arrow could be used to move left an entire word. Alternate-left- and -right-arrows could move the cursor to the appropriate end of the line.

Another extension could use the disk to preserve the contents of the string buffer between invocations of the Forth system. This could be done either with blocks, in the traditional Forth manner, or with file extensions to Forth.

As currently implemented, invoking the history capability overwrites whatever is in the edit buffer. One could rewrite the code to allow the user to enter part of a new line, and then recall (perhaps by insertion) a line from the string buffer.

The current implementation uses variables, making the code non-reentrant. To use this code in a multi-user system, the variable will have to be replaced with user variables. The FastForth optimizing compiler handles user variables, so there is no benefit other than readability to making pseudo-constants with the user variables. In both FastForth and traditional, indirect-threaded Forth, the user variable approach actually occu-

Listing Two. Parallel scalar vector multiplication.

```
fload figure1
30 CONSTANT SIZE                    \ DEF  SIZE = 30 :
USER VARIABLE U.I FORTH             \ VAR  U.I     ,
SIZE []WORD V1                      \      V1[SIZE],
SIZE []WORD V2                      \      V2[SIZE]:
SIZE 1+ []CHAN N                    \ CHAN N[SIZE+1],
SIZE    []CHAN W                    \      W[SIZE] ,
SIZE    []CHAN E                    \      E[SIZE] :
                                    \ VAR  T1,T2,T3,T4:

: VECT-MULT                         \ PROC VECT.MULT =
                                    \ SEQ
  SIZE 0 DO                         \   U.I = [0 FOR SIZE]
    I 1+ I V1 !                     \     V1[U.I] := U.I + 1
    I 1+ I V2 !                     \     V2[U.I] := U.I + 1
  LOOP                              \   PAR
  SIZE 0 DO I U.I !                 \     U.I = [0 FOR SIZE]
    I U.I !                         \       PAR
    ||( U.I @ V1 @ U.I @ W >C )     \         W[U.I] ! V1[U.I]
    ||( U.I @ V2 @ U.I @ E >C )     \         E[U.I] ! V2[U.I]
    ||(                             \         SEQ
      U.I @ W C>                    \           W[U.I]   ? T1
      U.I @ E C> *                  \           E[U.I]   ? T2
      U.I @ N C> +                  \           N[U.I]   ? T3
      U.I @ 1+ N >C                 \           N[U.I+1] ! T1*T2+T3
    )                               \
  LOOP                              \
  ||( 0 0 N >C )                    \     N[0] ! 0
  ||( SIZE N C> U. )                \     SEQ
                                    \       N[SIZE] ? T4
                                    \       WRITE(T4) :
  PAR ;
```

Listing Three. Parallel alarm clock.

```
fload figure1
\ Alarm clock
ANEW ALARM
HIDDEN ALSO EDITOR ALSO

USER
VARIABLE ALARM-HM
VARIABLE ALARM-STRING
FORTH

: ALARM" ( HOURS MINUTES | <TEXT>" )
  SWAP FLIP + 0 0. B>T DROP ALARM-HM ! \ Get time
  HERE ," ALARM-STRING !               \ Get alarming string
  EV(
    ALARM-HM @
    BEGIN                              \ Wait ...
      1000 FOR PAUSE NEXT              \ Skipping for efficiency
      DUP GETTIME DROP U<=
    UNTIL
    DROP SINGLE                        \ Disable parallelism
    SAVESCR SAVECURSOR                 \ Save screen & cursor
    DKGRAY >BG WHITE >FG               \ Select colours
    TRUE ALARM-STRING @ COUNT ?SOFTERROR       \ Alarm !!
    RESTCURSOR RESTSCR MULTI           \ Restore screen & cursor
  )                                    \ Enable parallelism
;
ONLY FORTH ALSO DEFINITIONS
```

pies less room in the dictionary than the variable approach.

Availability

In the best Forth tradition, the code is released to the public domain. Enjoy it in good health.

FastForth for the Atari ST, including the above code, may be had in alpha release from the author, Charles Curley, P.O. Box 2071, Gillette, Wyoming 82717-2071. Please consult the author for the current state of documentation, etc.

Readability Revisited

Garth Wilson
Whittier, California

I had been thinking about writing an article on readability of Forth code, but I became more determined after going through a recent issue of *Forth Dimensions*. It was full of things that reduce code readability, cover to cover. It's no wonder some have called Forth a write-only language! It doesn't need to be this way. If a programmer gets the application working but the code is unreadable, he hasn't done his job.

After looking through my library at others' treatments of the subject, a part I see as lacking special attention is what Jack Ganssle calls "pretty code."* It has not been stressed enough in our programming industry. I've had non-programming superiors who thought that any work beyond "just getting it working" was frivolous, and that the programmer who spent the time was not sensitive to the company's needs.

Jack Ganssle tells of a Volkswagen Beetle maintenance book he had which strongly encouraged the owner to keep the engine clean. One reason was that you're more likely to keep it in good shape if you can do the necessary maintenance without getting filthy. We've all seen a lot of "filthy engines" in our industry—ones we don't want to touch.

Producing readable code is not an objective procedure with an exact formula.

I've had code that was sold as "toolbox" code that was too unreadable to figure out what was necessary in order to use a small piece of it in my own application. I thought, "Why'd they even bother?" It doesn't have to be that way.

Special attention will be given here to layout, and to making the layout more perceptible. "Layout" here refers to the decisions as to what words will be put on a particular line of source code, where in the line they will start, use of blank lines to separate code "paragraphs," where the comments will go, etc. Let's look first at vertical alignment, then line breaks, zero-stack-effect lines, comments (including stack-effect), and case.

* "The C Blues," *Embedded Systems Programming*, April 1993

Vertical Alignment

Comprehension is accelerated dramatically by vertical alignment of related elements. Structure words need to be aligned vertically unless the structure is very simple; for example, `BEGIN ?TERMINAL UNTIL` would fit nicely on a single, very readable line; but `?DO` and `LOOP` in Listing One need to be one above the other, obviously standing out from the contents of the loop.

The indentation should be at least three spaces. Recently I was trying to read some code in a book. The indentation for setting off loops was only one space, which made it look like the printhead just didn't come back to the same place with every carriage return, producing a ragged margin.

Having many nested structures can be a pain, and can usually be avoided by making each nested level a separate definition. In some cases, however, it is difficult to come up with a truly descriptive colon-definition name that's any shorter than the code itself. Having neither good names nor the code right there, you will have trouble understanding the flow of what the code is supposed to do. Vertical alignment and readability can still be preserved using methods like the one shown in Listing Two-a instead of Listing Two-b.

Listing Two-b is far more confusing, and takes just as many lines, despite the lack of blank lines to set off the different parts. I've seen this kind of thing carried to the extreme, making a huge sideways 'V' from the top of the printed page to the bottom, with the point of the 'V' indented so far over to the right that it left no room for comments.

Many Forth programmers would say a definition should never get this long, and would proceed to "flesh it out" in Forth one-liners, requiring many, many levels of nesting at execution time. I would go along with this only as long as the colon-definition names are very descriptive of what they do, and that the "fleshing out" is not used as a substitute for good commenting or layout.

Vertical alignment of similar words on lines helps, too. Our brain homes in on patterns that simplify the mental processing and memory requirements of the material. Consider Listing Three-a. These simple definitions can be

Listing One. Vertical alignment and indenting.

```
: LPTTYPE          ( ADR CNT -- )     ( "TYPE" VERSION FOR PRINTER [LPT].  )
   OVER + SWAP                        ( type WILL CALL THIS IF OUTDEV=LPT. )
   ?DO                                ( LOOP LIM & INDX ARE ACTUAL ADDRs.  )
      I C@  LPTEMIT                   ( @ & PRINT CHR AT NXT ADR IN STRING.)
      OUTERR @                        ( SEE IF LPTEMIT HAD ANY PROBLEM.    )
      IF CONSOLE LEAVE THEN           ( IF LPTEMIT NOT SUCCESSFUL, SET FOR )
   LOOP                         ;     ( LCD OUTPUT AGAIN & LEAVE THE LOOP. )
```

Listing Two-a. When nested structures are necessary.

```
: SAMPLEWORD       ( input -- output )    ( comments comments...)

   CONDITIONS                  IF         ( comments comments...)
   ACTIONS ACTIONS ACTIONS                ( comments comments...)
   ACTIONS ACTIONS ACTIONS     ELSE       ( comments comments...)

   CONDITIONS                  IF         ( comments comments...)
   ACTIONS ACTIONS ACTIONS                ( comments comments...)
   ACTIONS ACTIONS ACTIONS     ELSE       ( comments comments...)

   CONDITIONS                  IF         ( comments comments...)
   ACTIONS ACTIONS ACTIONS                ( comments comments...)
   ACTIONS ACTIONS ACTIONS     ELSE       ( comments comments...)

   ACTIONS          THEN THEN THEN    ;
```

Listing Two-b. The scrambled-eggs version.

```
: SAMPLEWORD     ( input -- output )
   CONDITIONS
   IF ACTIONS ACTIONS ACTIONS
      ACTIONS ACTIONS ACTIONS
   ELSE CONDITIONS
      IF ACTIONS ACTIONS ACTIONS
         ACTIONS ACTIONS ACTIONS
      ELSE CONDITIONS
         IF ACTIONS ACTIONS ACTIONS
            ACTIONS ACTIONS ACTIONS
         ELSE
            ACTIONS
         THEN
      THEN
   THEN                  ;
```

made much more mentally manageable by simply lining things up so the brain can "factor" it. This example has many factors that the brain should only have to catch once, greatly simplifying the job of figuring out what someone else (or maybe you yourself) has written. Listing Three-b takes care of those.

Listing Four-a shows a colon definition I wrote to figure out which key or keys, if any, were being pressed on a 16-key keypad connected through port A (PA) of a 65C22 VIA (versatile interface adapter) IC. The four most significant bits were set up to be inputs, and the four least significant were to be outputs.

There's more description of it farther down. Listing Four-b shows the same code without the vertical alignment. It resembles a kitchen or bathroom you don't even want to be in (let alone use) because it's so dirty.

Something else crept into Listing Four-b—the ; is on the left margin. This type of thing is done in C a lot (with the curly braces); but when we see many colons and semicolons all on the margin, it takes more attention to tell

Listing Three-a.

```
: BOLDON 1B EMIT ." E" -2 LPT-OUT +! ; ( PUT PRINTER IN BOLD MODE. )
: BOLDOFF 1B EMIT ." F" -2 LPT-OUT +! ; ( TAKE PRINTER OUT OF BOLD. )
: TINYON F EMIT -1 LPT-OUT +! ; ( PUT PRINTER IN COMPRESSED.)
: TINYOFF 12 EMIT -1 LPT-OUT +! ; ( TAKE PRINTER OUT OF COMPRESSED.)
```

Listing Three-b. Pave the way to mental factoring.

```
: BOLDON  1B EMIT ." E" -2 LPT-OUT +!   ;  ( PUT PRINTER IN BOLD MODE.       )
: BOLDOFF 1B EMIT ." F" -2 LPT-OUT +!   ;  ( TAKE PRINTER OUT OF BOLD.       )
: TINYON  0F EMIT       -1 LPT-OUT +!   ;  ( PUT PRINTER IN COMPRESSED MODE.)
: TINYOFF 12 EMIT       -1 LPT-OUT +!   ;  ( TAKE PRINTER OUT OF COMPRESSED.)
```

Listing Four-a. Keyboard scan word, lined up and tidy.

```
: WHICHKEYS ( -- key_cell )          ( OUTPUT A CELL W/ A BIT SET 4 EA KEY HIT. )
   7 VIAPA C!                        ( MAKE ONLY KBD ROW 1 LO.                  )
     VIAPA C@  8 SHIFT 0FFF OR       ( PUT n ON STK W/ 0 IN BITS WHOS KEY IS DN.)
   B VIAPA C!                        ( MAKE ONLY KBD ROW 2 LO.                  )
     VIAPA C@  4 SHIFT F0FF OR AND   ( AND-IN ROW-2 BITS.                       )
   D VIAPA C!                        ( MAKE ONLY KBD ROW 3 LO.                  )
     VIAPA C@          FF0F OR AND   ( AND-IN ROW-3 BITS.                       )
   E VIAPA C!                        ( MAKE ONLY KBD ROW 4 LO.                  )
     VIAPA C@ -4 SHIFT FFF0 OR AND   ( AND-IN ROW-4 BITS.                       )
   NOT          ;                   ( MAKE EA BIT WHOS KEY IS HIT A 1, OTHRS 0.)
```

Listing Four-b. The dirty-kitchen version.

```
: WHICHKEYS    ( -- key_cell )    ( OUTPUT A CELL W/ A BIT SET 4 EA KEY HIT.)
  7 VIAPA C!            ( MK ONLY KBD ROW 1 LO.)
  VIAPA C@ 8 SHIFT 0FFF OR  ( PUT n ON STK W/ 0 N BITS WHOSE KEY IS DN.)
  B VIAPA C!                  ( MAKE ONLY KBD ROW 2 LO.    )
  VIAPA C@ 4 SHIFT F0FF OR AND      ( AND-IN ROW-2 BITS.)
  D VIAPA C!                   ( MAKE ONLY ROW 3 LO.)
  VIAPA C@ FF0F OR AND ( AND-IN ROW-3 BITS.)
  E VIAPA C!             ( MAKE ONLY ROW 4 LO.    )
  VIAPA C@ -4 SHIFT FFF0 OR AND     ( AND-IN ROW-4 BITS.)
  NOT              ( MAKE EA BIT WHOS KEY IS HIT A 1, OTHRS 0.)
;
```

them apart, since they look so much alike. Nothing encountered while the compiler is on should be on the left margin. If you follow this suggestion, you'll know the end of the colon definition by the next thing that's on the margin, since most lines encountered when STATE is OFF will start on the left margin with :, CREATE, (, inputs for CONSTANT, etc.

Whatever you do, please don't print your code using proportional spacing! That's a sure way to mess up vertical alignment, as well as complicate the reading process by giving us something that is very different from what we're used to seeing on our screens as we program!

Editors

You might have already noticed that these lines are more than 64 characters long. Screen files have their place and their advantages, but in many situations a flexible text editor works much better. I use the Norton programmers' text editor with three-button mouse support (but no GUI!) in a very simple system I have set up where I can develop embedded-system code *on* a target system (in its RAM). The PC is only used for keeping the source code and, later, for metacompiling to put finished code into ROM for the target.

In two or three seconds (the main limitation being typing speed), I can go from writing a small piece of code,

Listing Five. Include descriptive diagrams with a text editor.

```
(      COL COL COL COL COL COL COL COL COL COL COL COL COL COL COL COL
(       1   2   3   4   1   2   3   4   1   2   3   4   1   2   3   4   )
(      ||---------|---------|---------|---------||---------|---------||  )
(      ||   ROW 1 |   ROW 2 |   ROW 3 |   ROW 4 |                    ||  )
(      ||msb                                                      lsb||  )
(      ||<---------- K E Y C E L L   O N   S T A C K ---------->||  )
(
(
(                COL1   COL2   COL3   COL4
(                PA7    PA6    PA5    PA4
(             +------+------+------+------+
(   ROW1      | BIT  | BIT  | BIT  | BIT  |
(   PA3       |  15  |  14  |  13  |  12  |
(             +------+------+------+------+
(   ROW2      | BIT  | BIT  | BIT  | BIT  |
(   PA2       |  11  |  10  |  9   |  8   |
(             +------+------+------+------+
(   ROW3      | BIT  | BIT  | BIT  | BIT  |
(   PA1       |  7   |  6   |  5   |  4   |
(             +------+------+------+------+
(   ROW4      | BIT  | BIT  | BIT  | BIT  |
(   PA0       |  3   |  2   |  1   |  0   |
(             +------+------+------+------+
(
(      Keyboard front view, with row
(      and column labels.
(      Keyboard is a 4X4 matrix.
(      Normally all the rows are held
(      low, and when a key is pressed,
(      one of the columns gets pulled
(      down, which also pulls the IRQ
(      [CA1] down [true] to assert the
(      IRQ to tell the µP to see which
(      key was pressed.  Actually, we
(      are not using keypad interrupts
(      at this time, but the hardware
(      is there should we want to in
(      the future.
```

```
8000 CONSTANT    ^KEY (   up arrow
0800 CONSTANT    vKEY ( down arrow
0008 CONSTANT MENUKEY
0004 CONSTANT  YESKEY
0001 CONSTANT   NOKEY

CREATE KEY#TBL
  4000 , 1 C,      0100 , 6 C,
  2000 , 2 C,      0040 , 7 C,
  1000 , 3 C,      0020 , 8 C,
  0400 , 4 C,      0010 , 9 C,
  0200 , 5 C,      0002 , 0 C,
```

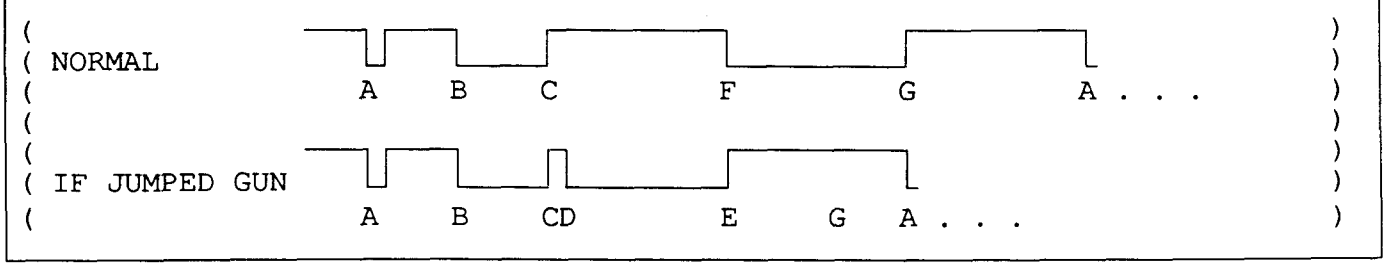
```
(   +------+------+------+------+
(   |  ^   |  1   |  2   |  3   |
(   +------+------+------+------+
(   |  v   |  4   |  5   |  6   |
(   +------+------+------+------+
(   |      |  7   |  8   |  9   |
(   +------+------+------+------+
(   |      |      |  0   |      |
(   | MENU | YES  |      |  NO  |
(   +------+------+------+------+
(          ENTER          EXIT
(        CONTINUE
```

to compiling it (if necessary), to trying it on the target—all without leaving, suspending, or even backgrounding the editor. It is set up so no code changes are needed to make the transition from RAM to ROM for the finished system. It's very simple, and the only hardware and software required has been out for years. It's a natural for Forth. (That will be material for another article. In spite of its simplicity, I haven't found anyone else who has heard of such a thing, so evidently it does deserve an article.)

Use a text editor where possible, so you have lots of room for comments, and so you can use more of the special characters and have short descriptive names like 100Ω, ℃, ½LIFE, or ±2%. Go ahead—use Å for angstroms. You can replace `(x^2+y^2)^1/2` with $\sqrt{(x^2+y^2)}$ without any of those hidden word processor control codes that are hostile toward compilers.

The only drawback is that you may have to modify `L>NAME` (or another Forth word that takes a CFA or LFA to find the NFA). This is because it typically backs up until it encounters the first byte whose high bit is set. This drawback is minor, since if you don't make the modification, the only effect you will probably notice is that `WORDS` or decompiling words will print the names with special characters wrong. Just in case the high bit is set in an actual character of the name itself, `L>NAME` needs to see if the length indicated by the five least-significant bits agrees

Listing Six. Even add timing diagrams with a text editor.



Listing Seven. Use of binary when individual bits are of interest.

```
: SYNCSERSETUP  ( -- )
                [B]
   VIADDRB C@          1 OR  VIADDRB C!     ( SET PB0 AS OUTPUT FOR STROBE LINE.  )
   VIAACR  C@ 00010100 OR                   ( ENABL SR TO SHIFT OUT UNDR T2 CTRL. )
              11010111 AND VIAACR  C!       ( DECR AT Φ2 RATE.                    )
              00100100     VIAIER  C!       ( DISABL SR-EMPTY & T2 IRQs.          )
                                            ( SET T2 FOR ABT 100kHz SHFT RATE, LO )
              [H]     8    VIAT2CL C!       ( BYT 1ST. T2 GETS RST AUTOMATICALLY  )
                      0    VIAT2CH C! ;     ( SO ITS NOT REALLY 1-SHOT IN SR MODE )
```

with the number of bytes already backed over. If there is disagreement, it has not arrived yet at the NFA, and it needs to keep looking. You might see right away that a few combinations could still fool it. Those are easy enough to determine and avoid.

A text editor will allow you to use the graphics characters for diagrams in your comments. The diagrams in Listing Five are out of an application I did a couple of years ago. They preceded my keyboard-scan code in Listing Four-a.

Where an explanation of timing is appropriate to understanding code, you can even include a timing diagram. For example, following a timing description in the comments on a recent project, I put in the diagrams in Listing Six to graphically summarize.

The source code should be more than just compiler-needed input—it should be the major software document. The diagrams are worthless to a compiler, but are priceless to someone who has to come along later and figure out how to update the code.

A text editor like the Norton Editor can also allow you to look at two files at once with a split screen, and to copy portions from one to the other, test portions for differences, etc. The transfer between files is transparent as you move the cursor with the mouse.

You can scroll. You can use block markers as book marks which the editor can find instantly after you have been looking at something elsewhere in the file. You can use different modes of search, or search-and-replace, including reverse and case-insensitive. The searches will be extremely fast, since usually the whole file is in RAM and disc access is unnecessary. You can flip the case from the cursor to the beginning or end of the line to transfer from upper- to lower-case (or vice versa) without retyping. You can reformat a paragraph, which is useful for comment paragraphs when you need to insert or delete something in the middle without messing up all the line lengths.

If you can forego the few advantages of screen files, a text editor can give you far more flexibility than the screen editors I've seen.

More Vertical Alignment

Listing Seven shows a colon definition I used to set up a synchronous serial port on a 65C22 VIA (versatile interface adapter) IC. The register names came right out of the data book, which has register diagrams showing the bit-by-bit functions. We want to pay attention to individual bits (not the overall number they make up), and we want to mask and set or clear individual bits; so it makes sense to use binary and line them up vertically, just as our second-grade teachers taught us to do for addition.

Sometimes, immediate do-nothing words are very helpful for clarifying code. You could have words like

```
: SAMPLES ; IMMEDIATE
: kHz     ; IMMEDIATE
```

to use in a sequence that plays back a specified set of samples at a specified rate, like

```
TBL1   2000 SAMPLES   8 kHz   PLAYBACK
```

Alternatively, `SAMPLES` could, for example, store the number on the top of the stack for `PLAYBACK` to use, so the net stack effect of `2000 SAMPLES` would be `( -- )`.

Now and then, we hear talk of programming languages or command structures that are very "English-like," as if it were a plus. Frankly, English is a lousy pattern for a programming language to follow. For technical communication, it's far better than some other languages, yet confusion still abounds.

Listing Eight-a. Maximum-stack-depth version—hard to understand or add comments.

```
: SETLCDADR ( n -- )
   F AND DUP 7
   > IF 7 AND 40
   OR THEN 80
   OR LCDINSTR! ;
```

Listing Eight-b. Sentence-line version.

```
: SETLCDADR ( n -- )          ( n IS CHR POSITION #, RANGE 0-F.          )
   F AND                      ( LIMIT LCD ADR TO 00-0F FOR 16 CHR.       )
   DUP 7 >                    ( SEE IF CHR ADDR IS FOR 2ND ½ OF DISP.    )
   IF 7 AND 40 OR THEN        ( IF SO, CLR BIT 3 & SET BIT 6.            )
   80 OR                      ( SET BIT 7 TO INDICATE LCD CHR ADR.       )
   LCDINSTR!            ;     ( GIVE ADR AS INSTRUCTION TO LCD.          )
```

Listing Nine.

```
: LCDDATA!   ( C -- )            ( LCD DATA STORE.                              )
               RIOTDRA C!        ( SET UP DATA LNs ON LCD.                      )
   LCDDATAWR   RIOTDRB C!        ( SET UP R/W & RS ON LCD.                      )
   LCDDATAWR+E RIOTDRB C!        ( SET LCD ENABL LN TRUE SO LCD TAKES BYT.      )
   LCDDATAWR   RIOTDRB C!        ( SET LCD ENABL LN FALSE AGN TO END WR CY.)
   LCDNOWR     RIOTDRB C!  ;     ( SET R/W LN BK TO RD.                         )
```

In spite of this, sometimes it is still practical to make an English sentence. For example, you can have

```
TURN OFF "TEST" LED
```

where TURN is a variable, OFF is a standard Forth word that zeroes the cell at the address shown by the number at the top of the stack, "TEST" is a constant with a one in the bit corresponding to the light-emitting diode (LED) labeled "TEST" on the front panel, and LED is a colon definition that examines TURN to know whether to turn the specified light off or on. There is no question left as to what this does.

Line Breaks

A common problem contributing to unreadability is that of poor word arrangement on lines. Prose would be very awkward done this way:

The manager called Betty.
He asked her to come into his office.
The manager dictated a letter.
Betty took the dictation.
Betty typed up the letter.

but this is perfect for programming! Violations will often be accompanied by very inadequate commenting, since it is harder to get a comment sentence to match up with a line of code that does not resemble a sentence. It is ironic that explanation is needed most when it's hardest to fit in.

I wish I could include some examples of code I've seen where these principles were severely violated but, if I did, someone would probably be very unhappy. So I just tried to come up with some artificial examples.

Listing Eight shows a definition used to tell a one-line, sixteen-character, liquid-crystal display (LCD) where to put the next character it receives. First look at the "sentence-line violation" version in Eight-a.

It becomes much more readable when we use the line breaks to divide program "sentences." This also makes it easier to match some good comments to each line, which in turn further improves human comprehension. Look at Eight-b.

LCDDATA! in Listing Nine is an example of the program sentence-line concept, coupled with vertical alignment that improves our mental "factoring." This word was used to store a data character (as opposed to an instruction byte) to the same LCD. The RIOT (RAM, I/O, and timer) was an IC on a board I was working with. DRA and DRB are its data registers A and B. The words in the left column are constants.

There is nothing left on the stack at the end of any of the lines in LCDDATA! in Listing Nine. This brings us to another point that ties in with the sentence-line concept.

Zero-Stack-Effect Lines

Code is much easier to understand if we minimize each line's net stack effect as well as the final stack depth. This is closely related to the sentence-line concept, in that we want to tie up loose ends as much as possible before finishing the line. More often than not, it is impossible to

Listing Ten-a. Zero-stack-effect line.

```
TMBUF ARCTM [ YR MINUTES - 1+ ]LIT CMOVE    ( Copy time, date to arc buffer.)
```

Listing Ten-b. A parameter calculated on each line.

```
CALCULATE SOURCE ADDRESS AT RUN TIME          ( comments comments comments  )
CALCULATE DESTINATION ADDRESS AT RUN TIME     ( comments comments comments  )
[ CALCULATE COUNT AT COMPILE TIME ]LIT        ( comments comments comments  )
CMOVE
```

Listing Eleven. Comments that need comments.

```
CODE CODE CODE CODE          ( ^ max addr addr' addr'' tok_len)
```

eliminate all net stack effect; but it is important that we work toward that goal, nevertheless.

A line with a CMOVE might simply be like Listing Ten-a. But if it takes a few operations to calculate the parameters, each of those operations will usually need its own line, as in Listing Ten-b.

Now (in listing Ten-b) we have a code paragraph made up of four sentence lines. If this is only part of a colon definition, a blank line should be used to separate it from other code that may come above and below it.

Stack-Effect Comments

When code can be written in minimal-stack-effect lines, there will be little need for stack comments for each line. Sometimes, however, things will get a bit confusing, and there may not be any good way to get around it.

If there is any question about what's left on the stack at the end of a line, that should be cleared up in the comments. In those cases, I like to start the comment line with a ^ as a stack symbol, followed by a description of what's on the stack after the line is finished executing.

Listing Eleven shows the stack effect comment following a line of code in something we bought. It hardly qualifies as a description. Maximum what? Addresses of what? Granted, when there are five things on the stack to tell about, you don't have much room to tell what each one is. But since that was the case, they should have put a few lines of "glossary" above or below the code, telling you that when you come upon "addr," understand that it means such-and-such. The same would apply at the next step up, where there is truly a description, but accomplishing it required non-standard abbreviations beyond what is intuitive.

Dropping down to use another line may initially seem like an obvious solution to the problem of insufficient comment room. However, if this requires putting comment space between lines of code that should not be separated, it may partially defeat the purpose.

Comments

Much has already been said about comments. I will only add that if there is not enough room to the right of the code to put in complete comments, by all means—put a paragraph or two (or whatever it takes) above the code to describe what it will be doing, and why. No abbreviations are necessary here. Make a complete description. Leave *nothing* hidden! Expose everything! Typing it in while it is fresh in your mind will take a fraction of the time it would take to figure it out with inadequate comments a year or two later when you want to make an update. It will be even more valuable if someone else has to figure it out.

In some cases, you may need to write about hardware or other limitations encountered that led up to the decision to do it the way you did. Sometimes when I didn't do this, those things have slipped my mind and I would think, "Hey, there's a much better way to do this!" After spending some time on it, I would start remembering that I had already tried my "better" idea and that there was a good reason for not doing it that way.

Case

Lower-case is generally only for internals, like `branch`, `lit`, `litq`, etc. For example, in the Forths I've used, DO compiles do (a different word), and CREATE compiles the address of the run-time routine called `create` (also a different word). The programmer rarely accesses the internals directly. Because of this, some Forths (LMI's UR/FORTH, for example) start out by putting the PC in "Caps Lock."

The ascenders and descenders of lower-case letters are part of what our brain uses to recognize words quickly when we read prose, when our eyes stop once for every few words as we read quickly. Our process of reading code, however, is much different. When we read prose, the line divisions are mostly meaningless, existing primarily because an 8.5" x 11" book is much more manageable than a ribbon a mile long. However, the line breaks in

(Continues on page 31.)

Print ZIP Barcodes

Walter J. Rottenkolber
Mariposa, California

Since the early 1980's, the U.S. Postal Service (USPS) has been making the change from manual to automated mail processing. Both optical character recognition (OCR) and POSTNET (Postal Numeric Encoding Technique) barcode scanning technology were implemented to guide the sorting and distribution machinery. OCR was designed to read the address and print a barcode at a time before zip codes were widely distributed.

I suspect that great hope was held for OCR technology. Letting the computer generate the zip code from the printed address would have eliminated the enormous task of disseminating zip code data. But even today, OCR can correctly scan only 40–50% of labels, whereas over 98% of barcodes are readable. As a result, the USPS has switched to the barcode as the primary zip-encoding method, and as of March 21, 1993 has implemented new barcode-reading equipment.

The original (conventional) barcode has very strict placement requirements. Its read area is a 5/8" x 4 3/4" block in the lower-right corner of the envelope. The barcode is centered in the area, with the base located 1/4" above the bottom of the envelope. The left end of the bar should be between 4 1/4" and 3 1/2" from the right edge of the envelope. The USPS still uses this area to print barcodes, so it should be left clear of any other printing.

The new equipment allows for a far wider placement of the POSTNET barcode. This area is between the envelope bottom and 4" above the bottom (including the conventional barcode area), and 1/2" inside the right and left sides of the envelope. However, the left-most end of the bar must be less than 10 1/2" from the right edge. The barcode can also be part of the address label, but it doesn't have to be.

In a label, the preferred location of the barcode is above the name field or any optional keyline or endorsement lines. This leaves the address fields clear for OCR, if necessary. However, it can be located at the bottom of the label. The main requirement is that at least a 1/25" gap separate the top and bottom of the bar from other writing, and that the right and left edges have at least a 1/8" (better 1/4") clearance from envelope window edges or other printing.

The zip code is encoded as a series of five-bit numbers. A 1 bit is represented by a tall bar, and a 0 bit by a short bar. The five bits are assigned the values 7, 4, 2, 1, 0. A list of the codes is given in Figure One. Numbers from zero to nine are coded by setting two bits which add up to the number (two-of-five code), For example, the number 3 is represented by setting bits one and two (2+1), and eight by bits four and one (7+1). The only exception is 0, formed by bits four and three, which would add up to eleven.

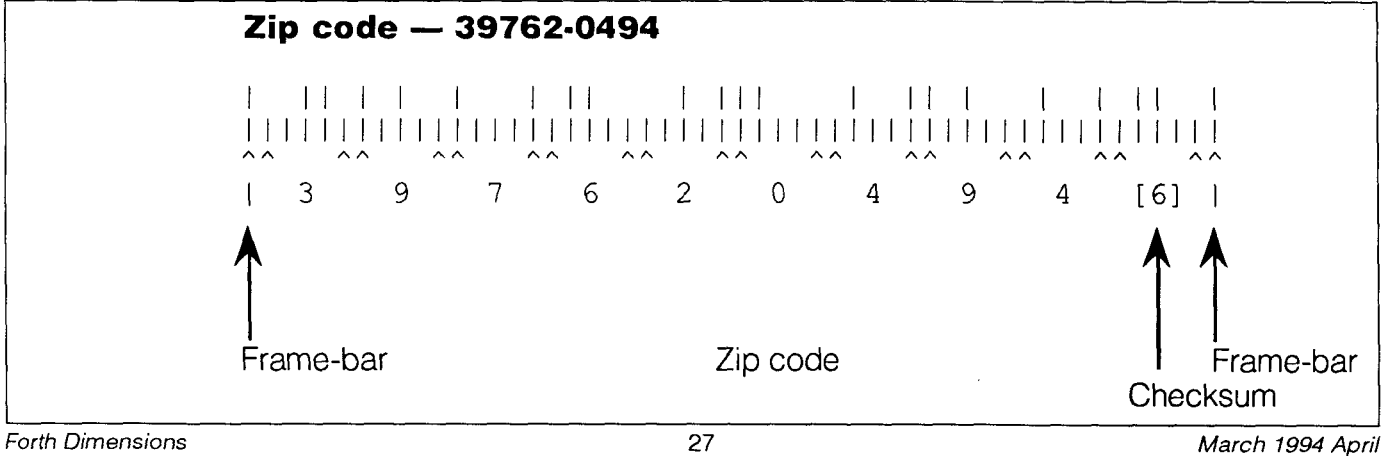
The POSTNET barcode is composed of three parts: *frame bars* at each end, the number code, and a correction character (Figure Two). The frame bar is simply a tall bar.

Individual mailers get faster delivery; businesses get cheaper rates.

Figure One. The two-of-five code.

11000	00011	00101	00110	01001
0	1	2	3	4
01010	01100	10001	10010	10100
5	6	7	8	9

Figure Two. Example barcode.



The number code is a five-bit translation of each digit in the ASCII number string. The correction character is a checksum, which is the ten's complement of the least significant digit of the sum of the digits in the zip code.

There are now three zip codes. The original, five-digit zip code (A field), the zip+4 code (C field), and the delivery point barcode (C-prime field, aka DPBC). This last is also known as the zip+6 code.

The first three numbers of the five-number zip code define a Sectional Region Center (SRC), which is part of a National Region defined by the first number. Zip codes start in Puerto Rico and the Virgin Islands with double zero, loop up to the New England states with a zero, and down the eastern seaboard to begin a lazy, snake-like meander westward. The nines are reached as it wings up the west coast to sweep past Alaska and end in the Pacific islands. Some less-populated islands have no zip codes. Depending on the population, the SRC could define a large city or several rural counties.

The last two numbers apply to up to one hundred Post Office Delivery Areas within a Sectional Region. This could be the post office itself, if applied to post office boxes.

The +4 code was added to narrow the sort to an address block. The first two numbers refer to the Delivery Sector, which marks several blocks or even a large office building. The remaining two define a Delivery Segment. This can be a specific block or side of a street, a floor in a building, or even a department in a large office.

The DPBC, implemented this year, adds two more digits (for a total of 11 digits) to the zip code. There digits are made up of the last two numbers of the street address, the post office box, or highway contract route box.

The DPBC is primarily an OCR or POSTNET code, and is not part of the usual zip code in the address label (unless it's an OCR label). The DPBC allows automated mail sorts within the address block, the last hundred address points. Since up to five hours of hand sorting is now required to get the mail ready for final distribution, the USPS has high hopes for the DPBC to ease this last bottleneck in the distribution system.

Printing the POSTNET barcode does not require a complex program once the zip code is in the form of a number string. In this barcode demonstration program, written in Forth, the zip code is entered either as an argument on the command line after the program word, or at a halt in the program (depending on which `ZIP$` is used).

The ASCII numbers are taken one-by-one from the string and converted to binary. Any non-digits are dropped, so formatting characters can be used to make entering the zip code easier. This digit is then used as an index into `ZBARCODE`, an array that holds the translation into the five-bit code. Since these codes are to be shifted to the left for the bits to be printed, I've placed them already shifted three bits to the left in the array byte. After the byte is fetched from the array, it is `FLIP`ped so that the encoded bits are at the significant end of the 16-bit integer, which is what will actually be shifted left. (Some Forths use ><

```
     1
0 .\ Zipcode Envelope/Label Maker
1
2 2 13 THRU
3 14  CAPACITY 1- INDEX
4 \S
5
6
7
8



     2
0 \ Zipcode Bar Print Routine
1
2     VARIABLE  APOS     4 APOS  !
3     VARIABLE  APOS#   40 APOS# !
4     VARIABLE  RPOS#    0 RPOS# !
5     VARIABLE  VTAB#    9 VTAB# !
6
7    DEFER  PSPACE
8 : 1PSPACE   ( -- )  0 (PRINT) ;
9 : 2PSPACE   ( -- )  1PSPACE  1PSPACE ; ' 2PSPACE IS PSPACE
10 : TALLBAR   ( -- )  127 (PRINT)  PSPACE ;
11 : SHORTBAR  ( -- )  112 (PRINT)  PSPACE ;
12 : FRAMEBAR  ( -- )  TALLBAR ;
13
14
15



     3
  \ Zipcode Bar Print Routine

  \ Printer Codes are for Okidata u92 Graphics Mode.
  : GRAFON  ( -- )   3 (PRINT) ;
  : GRAFOFF ( -- )   3 (PRINT) 2 (PRINT) ;

  : PCR     ( -- )  13 (PRINT) ;
  : PCRLF   ( -- )  PCR  10 (PRINT) ;
  \  Spaces to left edge of zipbar.
  : ZIPOS   ( -- )  PRINTING ON  APOS @ SPACES PRINTING OFF ;
  : STRTZIP  ( -- )  GRAFON ;
  : STOPZIP  ( -- )  GRAFOFF PCR ;



     4
  \ Zipcode Bar Print Routine

  \ Okidata 92 printer codes
  : PESC    ( -- )  27 (PRINT) ;
  : LETTER  ( -- )  PESC  49 (PRINT) ;
  : DRAFT   ( -- )  PESC  48 (PRINT) ;
  : 12CPI   ( -- )  28 (PRINT) ;
  : 10CPI   ( -- )  30 (PRINT) ;
  : 6LPI    ( -- )  PESC  54 (PRINT) ;
  : 8LPI    ( -- )  PESC  56 (PRINT) ;
  : PCANCEL  ( -- )  24 (PRINT) ;
```

```
     5
  \ Zipcode Bar Print Routine

  ( Set Binary )  2 BASE !
  \ Uses 5-bit code with 2 of 5 set, & bit values of 7 4 2 1 0.
  \   Code shifted 3 bits left.
  CREATE  ZBARCODE
      11000000 C,  00011000 C,  00101000 C,   \ 0,1,2
      00110000 C,  01001000 C,  01010000 C,   \ 3,4,5
      01100000 C,  10001000 C,  10010000 C,   \ 6,7,8
      10100000 C,                             \ 9
  DECIMAL


     6
0 \ Zipcode Bar Print Routine
1
2 : PZIP#  ( n -- )
3     ZBARCODE + C@ FLIP
4     5 0 DO
5         DUP 0< IF TALLBAR
6         ELSE SHORTBAR THEN 2*  \ Shift bit left
7     LOOP DROP ;
8
9 : DIGIT?  ( n -- f )  10 U< ; \ t= 0..9
10


     7
0 \ Zipcode Bar Print Routine
1
2 : ZIPBAR  ( a l -- n ) \ n= digit sum
3     0 -ROT BOUNDS  DO  I C@ 48 - DUP DIGIT?
4     IF DUP PZIP# + ELSE DROP THEN LOOP ;
5
6 : ZCHKSUM  ( n -- )  10 TUCK MOD - PZIP# ;
7 \  Checksum is 10's complement of least significant digit
8 \   in the sum of numbers in zipcode.
9
10 \ : ZIP$  ( -- a l )  PAD DUP 20 EXPECT SPAN @ ;
11 \  Used to enter zipcode as: ZIP (cr) zipstring (cr)
12
13 : ZIP$  ( -- a l )  BL PARSE-WORD ;
14 \  Used to enter zipcode on commandline: ZIP zipstring (cr)
15


     8
0 \ Zipcode Bar Print Routine
1
2 : PZIPBAR  ( a l -- )
3     FRAMEBAR  ZIPBAR  ZCHKSUM  FRAMEBAR ;
4
5 : ZIP1  ( a l -- )  \ Single print zipbar
6     ZIPOS  STRTZIP  PZIPBAR  STOPZIP ;
7
8 : ZIP2  ( a l -- )  \ Double print-pass zipbar
9     2DUP  ZIP1  ZIP1 ;
10
11 : ZIP  ( -- )  (S zipcode$ )
12 \ Enter Zipcode string on commandline; include leading zeros
13     ZIP$  DUP 5 < IF 2DROP EXIT THEN  \ min. 5 digits or quit
14     ['] 2PSPACE IS PSPACE  APOS# @ APOS !  ZIP2 ;
15               \ Set to ZIP1 for single print-pass zipbar.
```

as the flip word.)

The checksum is calculated by doing a `10 MOD` on the sum, which isolates the least-significant digit. Subtracting this number from ten gets the ten's complement. This value is then tacked onto the end of the barcode.

Printing the POSTNET barcode requires the printer's graphics mode. The printer-specific codes in the source listing—fed to the printer by `(PRINT)`—are for the Okidata u92. You will need to dredge up the codes and routines for your printer if it is another make.

To enhance the usefulness of the zip code routine, I added simple programs to print addresses on envelopes (`ENV`) and on labels (`RLBL` and `ALBL`). The label routines are set for three-across, 15/16" x 2 1/2" labels. `LZIP1` is complicated because the offset (`ZIPOS`) requires `12CPI` to set the column position, while the barcode needs `10CPI` to get better spacing. The label routine is quite simple, printing the addresses vertically, rather than across. Column position (one through three) is determined by `SETCOL` before you start printing.

The physical dimensions of the barcode are summarized in Figure Three. These are based on the original inkjet printers used by the postal service. The new scanners are supposed to have wider latitude, but the postal service hadn't updated the standards when I last checked. In theory, my Okidata u92 prints bars slightly too narrow, and tall bars slightly too short, according to the standards; but only testing will tell if the barcode scans.

The light reflected from the background must be at least 30% higher than from the barcode. Originally, this had to be in the red and green portions of the spectrum, but the equipment is now blue-sensitive (except to a narrow range of light blue—the kind found on graph paper). Generally, light colors are fine. Gray is a tricky color, since it appears lighter than its reflectivity would suggest. Dark red and dark green, the colors of Christmas-time, cause no end of trouble. A white label is, of course, ideal. Background patterns or writing that "shows through" should have less than a 15% print contrast ratio (PCR).

Similar considerations apply to the color of the barcode itself, which has to be 30% less reflective than the background. The scanner is partially infrared-sensitive, so a dark red bar should be avoided.

The postal service knows that mail sorting represents the area with the greatest potential for saving labor and money. At present, the primary encoding method is the POSTNET barcode. The equipment first scans for the barcode. If that fails, OCR is attempted and, lastly, human intervention.

To encourage pre-barcoding mail, the postal service is trying the carrot trick. For the individual, speedier service is promised. Since every time a letter drops out of the scan sequence it goes to the end of the line, mail processing is faster with a proper barcode in place.

Businesses are given price breaks for batch mailings, based on volume, length of zip code, and amount of pre-sorting. This includes first-class mail, not just "junk" mail. Mailing as few as 250 letters at one time could qualify for a discount.

A great advantage of pre-barcoding is that you get the savings without the hassle of pre-sorting. One thousand letters that are five-digit pre-barcoded, without sorting, will cost nine dollars less to mail than zip+4 pre-sorted ($0.233 vs. $0.242, first class). The longer (DPBC) barcode will get you even more savings.

Some of the new barcode readers can also handle flats, i.e., large envelopes and magazines. Whether pre-barcoded batch discounts will be extended to these isn't settled yet, but publishers should keep in touch with the postal service.

Getting the zip code information can be a problem. The five-digit code is widely distributed, but the +4 code takes a bit more doing. The +4 code for California alone takes four volumes, making a stack twice as high as the national codes. It's obvious that looking them up from printed references works only for an individual with a short mailing list.

For businesses, it's a job for computers. The USPS will start you off with a free, one-time search of the zip codes with data from your address files. After that, third parties specializing in zip codes will upgrade your label database for a fee. CD-ROMs with nationwide zip+4 codes are also available, if you have the capability of doing it yourself.

To make implementing pre-barcoded mail less painful, the postal service has published informative booklets explaining all.

For help in designing mail to work with barcode scanners and in upgrading your databases, the USPS has Mailpiece Design Analysts and Business Consultants available at their district centers. You can reach one via mail or toll-free telephone. The analyst I spoke with was eager to help and knew her zip code technology. The USPS is trying harder.

Figure Three. Barcode dimensions.

Bar width	0.015" to 0.025"
Tall bar height	0.115" to 0.135"
Short bar height	0.040" to 0.060"
Space between bars	0.012" to 0.040"
Pitch	22 ± 2 bars/inch

Dots should touch, but can be up to 0.005" apart.
Bar rotation (leaning) or pattern skew (tilt): less than 5 degrees total.

Length of code	minimum		maximum
five-digit	1.245"	to	1.625"
nine-digit	2.075"	to	2.625"
eleven-digit	2.495"	to	3.125"

```
   9
\ Return Address

: RETADDR  ( -- a1 a1 a1 a1 n )
    " Your Name"
    " Your Street Address"
    " Your Hometown, CA"
    " Your-Zipcode" 4 ;

\S
Enclose each line of address block in quote marks (").
Last line MUST be zipcode. Number is # of lines.


   10
\ Envelope Address

: PLINE  ( a l -- )  APOS @ SPACES TYPE ;

: (ADDR)  ( addr-blk... -- )  \ Stack has addr len of addr $'s
    PRINTING ON
    3 SWAP 2* 1- DO
        CR I ROLL I ROLL PLINE
    -2 +LOOP SPACE TYPE ( zip ) CR
    PRINTING OFF ;


   11
\ Envelope Address

: PRADDR  ( -- )
    RPOS# @  APOS ! RETADDR (ADDR) ;

: PADDR  ( addr-blk... -- )
    APOS# @ APOS !
    )R  2DUP ZIP2  R) (ADDR) ;

: VTAB  ( -- )
    PRINTING ON  VTAB# @ 0 DO CR LOOP PRINTING OFF ;

: ENV  ( addr-blk... -- )
    9 VTAB# !  0 RPOS# !  40 APOS# ! ['] 2PSPACE IS PSPACE
    LETTER  PRADDR  VTAB  PADDR  PCANCEL ;


    12
0 \ Label Maker
1
2 : SETCOL  ( n -- ) \ Col 1..3
3     DUP 1 3 BETWEEN NOT IF DROP 3 THEN
4     1- 32 * 4 + APOS ! ;
5
6 : SETLABEL  ( addr-blk -- )
7     12CPI 8LPI  LETTER  8 ( lpi )  OVER - VTAB# ! ;
8
9 : PLZIP  ( a l -- )  \ Single print zipbar
10    STRTZIP  PZIPBAR  STOPZIP ;
11
12 : LZIP1  ( addr-blk -- )
13    12CPI ZIPOS  10CPI  )R  2DUP PLZIP  R) ;
14
15
```

```
    13
0 \ Label Maker
1
2 : LZIP2  ( addr-blk -- )
3    LZIP1  LZIP1 ;  \ Double print zipbar
4
5 : DOZIP ( addr-blk -- )
6    8LPI ['] 1PSPACE IS PSPACE  LZIP2 ;
7
8 \  Note: Set column (SETCOL) to n= 1..3 before printing labels.
9
10 : RLBL  ( -- )  \ Return address label
11    RETADDR  SETLABEL  (ADDR)  VTAB  PCANCEL ;
12
13 : ALBL  ( addr-blk -- ) \ Mailing address label
14    DOZIP  SETLABEL  (ADDR)  VTAB  PCANCEL ;
15

    14
0 \ FIG  FDIM  Address
1
2 : FIG  ( -- n... )
3    " Forth Interest Group"
4    " P.O. Box 2154"
5    " Oakland, CA"
6    " 94621-2154" 4 ;
7
8 : FDIM  ( -- n... )
9    " Marlin Ouverson, Editor"
10   FIG 1+ ;
11
```


(Readability, continued from page 26.)

program code are *very* significant, and the ascenders and descenders of lower-case letters make for jagged lines that blur those boundaries.

Sometimes we have to use many abbreviations and acronyms to get a substantial comment to fit on a line, since if the comment does not say enough, it may be as nebulous as the code. Abbreviations and acronyms don't go over as well in lower-case, since we tend to try to pronounce them as written instead of recognizing their meaning. Even when there is plenty of room, I usually use the same abbreviations and acronyms, just to be consistent. However, in cases where you can forego these condensations, using lower-case for the comments can help the eye separate them from the code.

Remember that consistency also contributes to "pretty code" and helps readability. The numbers 0–9 are always "capitals"—why mix them with lower-case a–f in heX nUmBErS? Many systems will not even recognize lower-case letters as numbers, so using upper-case will improve portability, too.

Conclusions

Forth gives an incredible measure of freedom to the programmer. Since one of the implications is that code does not have to be readable to run, this flexibility is sometimes wrongly interpreted as a weakness. In an effort to make code more maintainable, some companies have set style standards that their programmers must meet. These bring up the bottom end, but also have the undesirable side effect of preventing use of some of the best strategies.

Producing neat, attractive, readable code ("pretty code") is not an objective procedure with an exact formula. There is an art to it. In a way, it's like penmanship; I don't expect everyone's to look just like mine, but if a person's writing is illegible, that is definitely a problem. The continual aim to achieve greater degrees of readability should be part of our Forth way of life. I hope this article has helped and encouraged in that direction.

ADVERTISERS INDEX

The Computer Journal 31

Forth Interest Group ..... centerfold

Harvard Softworks 32

Laboratory Microsystems, Inc. 16

Miller Microcomputer Services 5

Silicon Composers 2

HARVARD SOFTWORKS

NUMBER ONE IN FORTH INNOVATION

(513) 748-0390 P.O. Box 69, Springboro, OH 45066

Just how good is HS/FORTH? Well, it's already good enough to control mile long irrigation arms to water the nations crops, good enough to control orbiting shuttle experiments, good enough to analyze the nation's blood supply and to control the telephone switching systems. It monitors pollution (nuclear and conventional) and simulates growth and decline of populations. It's good enough to simulate and control giant diesel generator engines and super cooled magnet arrays for particle accelerators. In the army and in the navy, at small clinics and large hospitals, even in the National Archives, HS/FORTH helps control equipment and manage data. It's good enough to control leading edge kinetic art, and even run light shows in New York's Metropolitan Museum of Art. Good enough to form the foundation of several of today's most innovative games (educational and just for fun), especially those with animation and mini-movies. If you've been zapping Romulans, governing nations, airports or train stations, or just learning to type - you may have been using HS/FORTH.

Our customers come from all walks of life. Doctors, lawyers and Indian Chiefs, astronomers and physicists, professional programmers and dedicated amateurs, students and retirees, engineers and hobbyists, soldiers and environmentalists, churches and social clubs. HS/FORTH was easy enough for all to learn, powerful enough to provide solutions, compact enough to fit on increasingly crowded disks. Give us a chance to help you too!

You can run HS/FORTH under DOS or **Microsoft Windows** in text and/or graphics windows with various icons and pif files for each. What I really like is cranking up the font size so I can still see the characters no matter how late it is. Now that's useful. There are few limits to program size since large programs simply grow into additional segments or even out onto disk. The Tools & Toys disk includes a complete mouse interface with menu support in both text and graphics modes. With HS/FORTH, one .EXE file and a collection of text files are all that you ever need. Since HS/FORTH compiles to executable code faster than most languages link, there is no need for wasteful, confusing intermediate file clutter.

HS/FORTH runs under MSDOS or PCDOS, or from ROM. Each level includes **all** features of lower ones. Level upgrades: $25. plus price difference between levels. Source code is in ordinary ASCII text files.

HS/FORTH supports megabyte and larger programs & data, and runs as fast as 64k limited Forths, even without automatic optimization -- which accelerates to near assembler language speed. Optimizer, assembler, and tools can load transiently. Resize segments, redefine words, eliminate headers without recompiling. Compile 79 and 83 Standard plus F83 programs.

PERSONAL LEVEL **$299.**

Fast direct to video memory text & scaled/clipped/windowed graphics in bit blit windows, mono, cga, ega, vga, all ellipsoids, splines, bezier curves, arcs, turtles; lightning fast pattern drawing even with irregular boundaries; powerful parsing, formatting, file and device I/O; DOS shells; interrupt handlers; call high level Forth from interrupts; single step trace, decompiler; music; compile 40,000 lines per minute, stacks; file search paths; format to strings. software floating point, trig, transcendental, 18 digit integer & scaled integer math; vars: A B * IS C compiles to 4 words, 1..4 dimension var arrays; **automatic optimizer delivers machine code speed.**

PROFESSIONAL LEVEL **$399.**

hardware floating point - data structures for all data types from simple thru complex 4D var arrays - operations complete thru complex hyperbolics; turnkey, seal; interactive dynamic linker for foreign subroutine libraries; round robin & interrupt driven multitaskers; dynamic string manager; file blocks, sector mapped blocks; x86&7 assemblers.

PRODUCTION LEVEL **$499.**

Metacompiler: DOS/ROM/direct/indirect; threaded systems start at 200 bytes, Forth cores from 2 kbytes; C data structures & struct+ compiler; MetaGraphics TurboWindow-C library, 200 graphic/window functions, PostScript style line attributes & fonts, viewports.

ONLINE GLOSSARY **$ 45.**

PROFESSIONAL and PRODUCTION LEVEL EXTENSIONS:

FOOPS+ with multiple inheritance **$ 79.**

TOOLS & TOYS DISK **$ 79.**

286FORTH or **386FORTH** **$299.**

16 Megabyte physical address space or gigabyte virtual for programs and data; DOS & BIOS fully and freely available; 32 bit address/operand range with 386.

ROMULUS HS/FORTH from ROM **$ 99.**

Shipping/system: US: $9. Canada: $21. foreign: $49. We accept MC, VISA, & AmEx

Optimization of '386 Assembly-Language Code

David M. Sanders
San Francisco, California

I. Introduction

Many existing Forth compilers generate the code within a compiled word in the form of a sequence of pointers to other words. An "inner" interpreter then fetches these pointers, one at a time, and executes the appropriate routines that, when taken together, simulate the action of the Forth machine model. However, an increasing number of Forth implementations generate machine code directly during the compilation of Forth words. While the resulting code tends to take more space, it executes considerably faster. There is no coded "inner interpreter" in this type of Forth implementation; the inner interpreter is the processor's silicon-encoded, instruction-fetching mechanism.

This article discusses certain optimization techniques for machine code generated by a Forth compiler. The Intel 386 processor is used to illustrate the techniques, but many of them are applicable to a wide variety of processors, including the Motorola 68000 family. The discussion and examples are based upon 386 assembly language[1]; in practice, the optimizations would probably be carried out directly upon machine-code instruction codes, without the use of an assembler.

"Local optimizations" that are particularly applicable to Forth compilers can greatly reduce the amount of machine code generated.

In this article, I focus principally on techniques that are particularly applicable to Forth compilers, rather than on techniques that are common to compilers in general. The techniques discussed relate to certain "local" optimizations that, when employed together, can significantly reduce the amount of machine code that would otherwise be generated. In particular, I look at optimizations related to three areas:

1. optimizations related to buffering the top of the data stack in registers;
2. optimizations related to updating the data-stack pointer; and
3. optimizations related to combining generated code instruction sequences into shorter and faster code instructions.

I also look into some ramifications for Forth itself, in the form of some extensions.

II. Buffering the Stack Top in Registers

The 386 has a subroutine stack, which would be used by Forth as the return stack. When a machine-code-compiled Forth word calls another compiled word, an assembly-language CALL instruction is generated; the compilation of an exit from a Forth word would generate an assembly-language RET.

The 386 has no explicit stacks aside from the subroutine stack; the Forth data stack would need to be implemented by using a register such as EDI as the stack-pointer register, and incrementing and/or decrementing the pointer register as necessary.[2]

Ideally, the topmost part of the data stack should be buffered in one or more registers, so that the number of memory operations needed to manipulate the stack can be reduced. The number of memory operations can be further reduced if a variable number of items from the top of the data stack can be buffered in registers. Such a reduction in the number of generated memory accesses comes at the expense of compiler complexity. In this case, the stack-pointer register (EDI is used in the examples in this article) points to the top of the in-memory portion of the data stack.

During compilation, the compiler must itself keep track of the stack "state," which is used to indicate the registers that currently contain items from the top of the data stack. For the 386, the ideal registers to use for that purpose are

[1] For those of you not familiar with Intel assembler notation, the destination operand comes *before* the source operand. For example:
MOV EDX, EAX
This instruction sets EDX (the destination) to the contents of EAX (the source). In addition, when a register is used as a *base* or *index* register, the register name is enclosed in square brackets [ ]. As an additional example:
MOV [EDI]+4, EAX
This instruction sets the memory location addressed by [contents of EDI plus four] to the contents of EAX. Most instructions permit a memory location to be used as either a destination or a source.

[2] On the Intel 8088, which was used in early IBM PC's, a significant speed-up in instruction execution time could often be realized by using string instructions such as STOS, even without the REP instruction prefix, whenever possible. However, in designing the 386, Intel deliberately optimized instructions such as MOV [EDI], EAX so that any speed-up resulting from using string instructions without the REP prefix is completely negated, even if the pointer register (EDI) must be incremented or decremented in a separate instruction.

EAX, EBX, and EDX. This leads to a total of 16 possible stack states,[3] which are listed in Table One. Three of those states are illustrated in Figure One.

The stack state determines the code that is generated for each simple Forth word. ("Simple" words are identified by the settings of appropriate flags associated with the entries for those words. Simple words include dup, `drop`, `swap`, `+`, `and`, `xor`, `@`, `!`, `=`, `<`, `>`, and the like. Also, instructions such as `if`, `else`, `then`, `begin`, and `repeat` will, at compile time, cause appropriate machine code to be generated; such code is also dependent on the stack state.) Table Two shows examples of code generated for five different simple Forth words, using states 0, 1, and 12 (as shown in Figure One) as the starting states. Figures Two-a through Two-d illustrate four examples of stack-state changes that result from the compilation of words. Observe that some entries in Table Two generate no code for certain stack states; their actions are emulated by changes in the stack state at compile time.

Note that when a word does *not* cause code to be generated, it can still change the state of the stack during compilation. Since the changed stack state affects the code that is generated by subsequent words, the semantics of the original word are still preserved, in spite of the fact that no code was generated for that word. Figure Three illustrates this, by comparing the code generated by `drop swap` with the code generated by `swap` alone, in each case starting from stack state 1. While `drop` does not generate any code, it still changes the stack state and, thus, the code generated for `swap`.

Table One. Stack-top states by register usage.

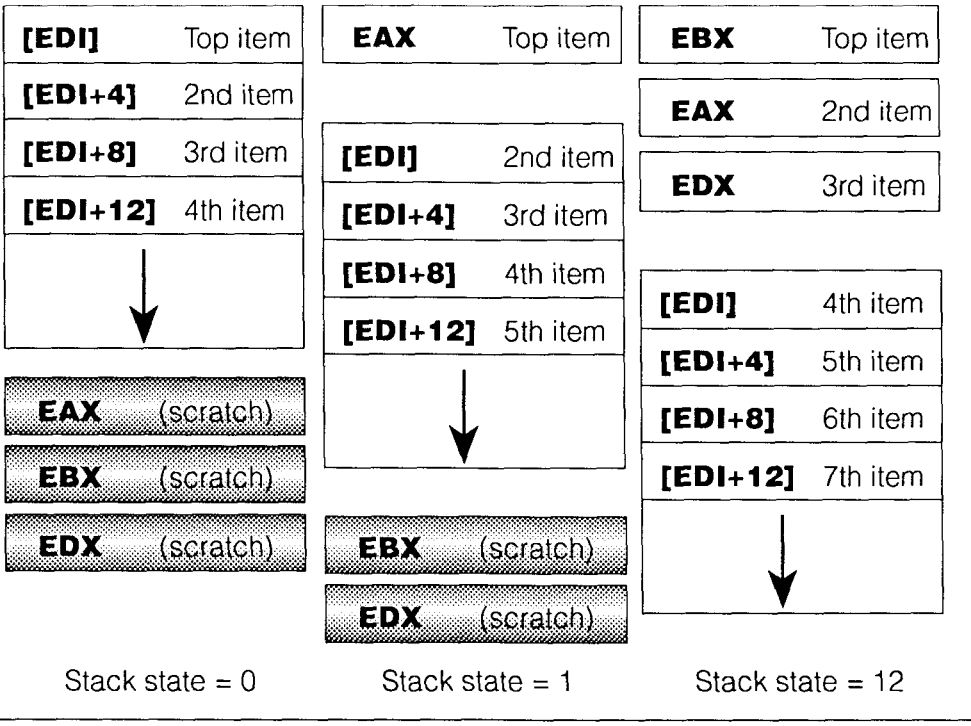
Stack state code	EAX contents	EBX contents	EDX contents
0	<nothing>	<nothing>	<nothing>
1	**Top item**	<nothing>	<nothing>
2	<nothing>	**Top item**	<nothing>
3	<nothing>	<nothing>	**Top item**
4	**Top item**	**2nd item**	<nothing>
5	**Top item**	<nothing>	**2nd item**
6	**2nd item**	**Top item**	<nothing>
7	<nothing>	**Top item**	**2nd item**
8	**2nd item**	<nothing>	**Top item**
9	<nothing>	**2nd item**	**Top item**
10	**Top item**	**2nd item**	**3rd item**
11	**Top item**	**3rd item**	**2nd item**
12	**2nd item**	**Top item**	**3rd item**
13	**3rd item**	**Top item**	**2nd item**
14	**2nd item**	**3rd item**	**Top item**
15	**3rd item**	**2nd item**	**Top item**

III. Updating the Data Stack Pointer

In some processors, such as those in the Motorola 68000 family, a register may be used for an address and also modified (post-incremented or pre-decremented) in the same instruction. However, the Intel 386 and many other processors don't incorporate this capability, or else restrict it to a few special instructions; the modification of a pointer must be performed in a separate instruction. Instead, the Intel 386 may add an offset value to a register to compute an effective address, with little or no penalty in execution time.

For the Intel 386, it makes sense if the register being used, for the data-stack pointer is not updated more often than absolutely necessary in the generated code. Instead, logical changes to the pointer can be tracked in a *virtual offset*. The virtual offset can then be applied as an offset in address calculations. When it is necessary to update the actual pointer value, the virtual offset is added as a literal to the pointer; the offset is then

Figure One. Examples of stack states.



[3] For a processor such as a member of the Motorola 68000 family, there are eight available data registers. Other processors may have a varying number of registers available for use as data-stack-buffering registers. However, as more registers are used to buffer the top of the stack, the number of stack states increases rapidly. For two registers, the number of states is only five. However, for four registers, the number of states increases to 65. For five registers, the number of states totals 326; for six registers, 1957!

If more than three registers are used to buffer the top of the data stack, a more "restricted" scheme for assigning registers to stack items is recommended in order to keep the total number of stack states to a manageable number. A more complete discussion of this topic, including possible register assignment strategies, is beyond the scope of this article.

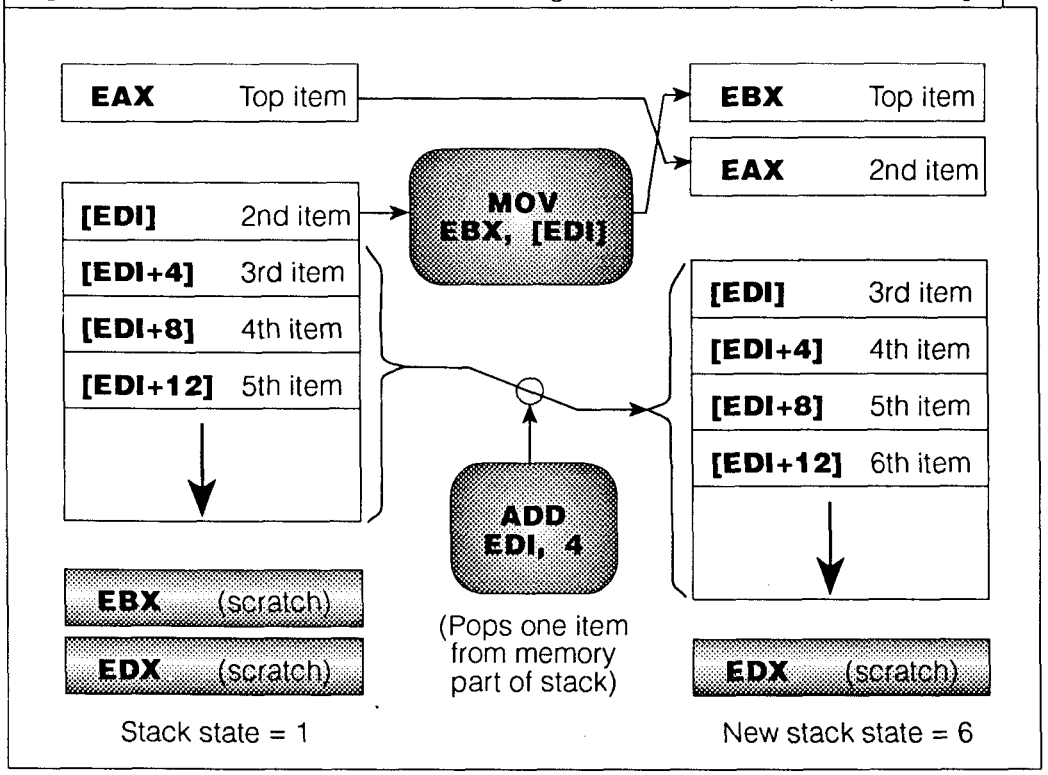
Table Two. Code generated for five words, and associated stack-state changes.

dup (state = 0)	**MOV EAX, [EDI]**	(new state = 1)
dup (state = 1)	**MOV EBX, EAX**	(new state = 4)
dup (state = 12)	**SUB EDI, 4**	
	MOV [EDI], EBX	
	MOV EDX, EBX	(new state = 13)
drop (state = 0)	**ADD EDI, 4**	(state unchanged)
drop (state = 1)	<no code generated>	(new state = 0)
drop (state = 12)	<no code generated>	(new state = 5)
swap (state = 0)	**MOV EAX, [EDI]**	
	MOV EBX, [EDI]+4	
	ADD EDI, 8	(new state = 6)
swap (state = 1)	**MOV EBX, [EDI]**	
(illustrated in Figure Two-a)	**ADD EDI, 4**	(new state = 6)
swap (state = 12)	<no code generated>	(new state = 10)
(illustrated in Figure Two-b)		
rot (state = 0)	**MOV EAX, [EDI]**	
	MOV EBX, [EDI]+4	
	MOV EDX, [EDI]+8	
	ADD EDI, 12	(new state = 14)
rot (state = 1)	**MOV EBX, [EDI]**	
	MOV EDX, [EDI]+4	
	ADD EDI, 8	(new state = 14)
rot (state = 12)	<no code generated>	(new state = 15)
xor (state = 0)	**MOV EAX, [EDI]**	
	XOR EAX, [EDI]+4	
	ADD EDI, 8	(new state = 1)
xor (state = 1)	**XOR EAX, [EDI]**	
(illustrated in Figure Two-c)	**ADD EDI, 4**	(state unchanged)
xor (state = 12)	**XOR EAX, EBX**	(new state = 5)
(illustrated in Figure Two-d)		

reset to zero. Figure Four illustrates the virtual offset and its use.

In the previous section, updates to the stack pointer (EDI) were generated in code for the sake of clarity. However, in practice, equivalent modifications are made to the virtual offset at compile time, and the actual offsets used in address calculations (using the [EDI]+*n* address mode) are modified by the prior value of the virtual offset. Table Three shows the same generated code as Table Two, but modified to accommodate a compile-time virtual offset; code for the `exit` word has been added to the table, in order to illustrate the code that is generated when the pointer register must be updated. (For all compiled Forth words, I am assuming that stack state 1 is the state for all entries and exits; the virtual offset must be *zero* for all entries and exits.)

Figure Two-a. Generated code and change-of-stack-state example for `swap`.



IV. Special Considerations for Stack-Top Optimizations

The optimization techniques for the data stack require special handling when program branches and/or direct accesses to the data stack are present. There are three different aspects that must be considered when optimizing a Forth program with these techniques:

1. When branch targets (and the sequential flow of execution) come together at a common point, the stack states and the values of *Voffs* (the "virtual offset" for the data stack) must be "rectified" so that a single, common stack state and Voffs value can be assumed by the compiler at that point, regardless of the source of each branch. This consideration applies both to

forward branches (such as those generated by the `if ... then` control structure) and to backward branches (such as loops). Figure Five illustrates several examples of rectifying the stack state and the value of Voffs at branch targets.

2. All compiled Forth words must be able to assume that the stack state and Voffs value are set to default values upon entry. This means that a calling word must enforce those defaults before calling another compiled word. Likewise, a calling word needs to be able to assume that those same defaults are present on return; the called word must enforce the defaults upon exiting. Coercion to defaults does *not* imply a restriction on the usage of the data stack as seen by the Forth programmer; it does, however, imply that extra code to set up the stack state and the value may need to be generated.[4]
3. If the stack pointer needs to be accessed (via the word `@sp` or a similar mechanism), it must be derived by taking the value of the data-stack pointer EDI, then modifying it to account for both the stack state and the current value of Voffs. Likewise, if it is expected that items on the stack will be made available for memory accesses, then the contents of all registers that buffer data-stack items (EAX/EBX/EDX) must be forced out to memory prior to any such accesses; this can be accomplished by rectifying the stack to state 0.

Figure Two-b. `swap`'s generated code and change-of-stack-state.

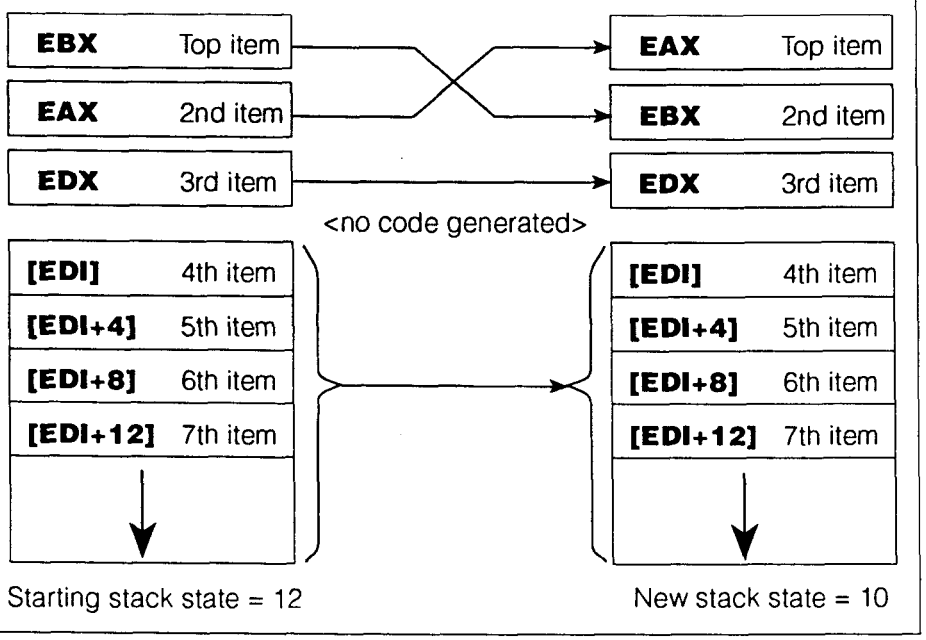
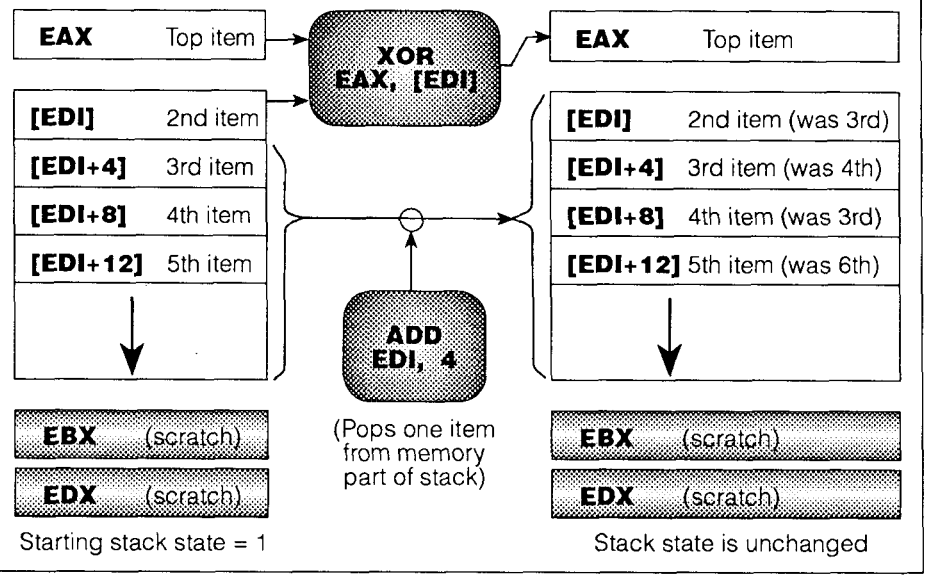


Figure Two-c. `xor`'s generated code and change-of-stack-state.



The rectification of one stack state to another is accompanied by the generation of code that moves values between the EAX/EBX/EDX registers and, possibly, the memory portion of the data stack; the value of Voffs may be modified in the process. Rectification of the value of Voffs to another value $Voffs_{(new)}$ involves generating code that adds the difference $Voffs_{(current)} - Voffs_{(new)}$ to the data-stack pointer EDI, then setting Voffs to $Voffs_{(new)}$. If rectification does not involve a change to either the current stack state or the Voffs value, no code is generated.

It is important to remember that both the stack state and the Voffs value are significant only during compilation. While the resulting generated code will reflect the influences of those values, there is no run-time tracking of what those values were in the generated code. (However, any data structures generated for debugging purposes will need to track those values in order to display the data stack's contents correctly.)

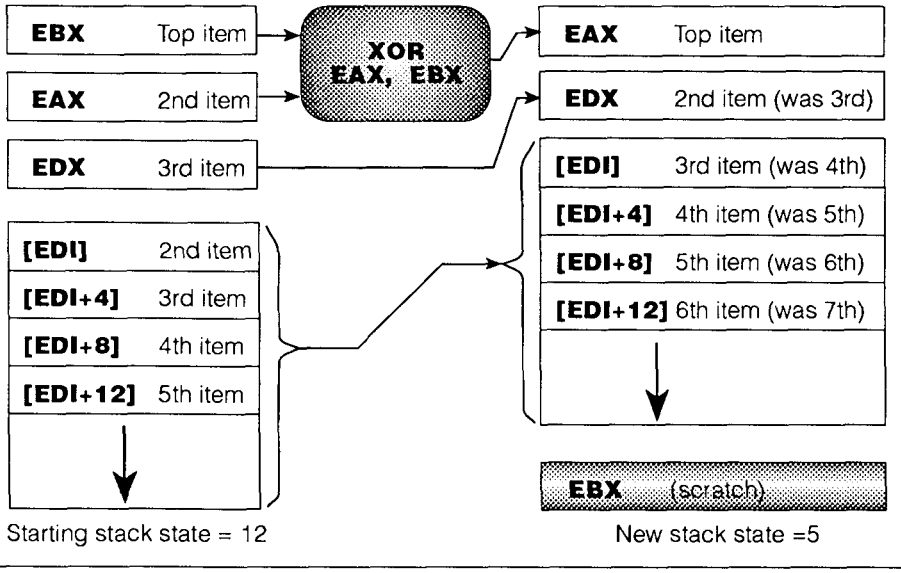
[4] The same considerations for calling a compiled Forth word also apply to the use of `execute` to call a word indirectly via an executable address on the data stack. In addition, Forth code should *never* attempt to enter any compiled Forth word, except at the legitimate entry point at the start of the word.

Some early Forth programs take advantage of the fact that many Forth implementations compile each new word in the form of a sequence of addresses that point to the called words. Since an increasing number of Forth implementations generate machine code directly, such techniques should be completely avoided. Instead, if a table of executable-word addresses is needed, the table should be built explicitly as an array of addresses that can be used with the `execute` word. (The ANSI Forth standard recognizes that different Forth implementations may use completely different methods to generate and interpret run-time code; it specifically prohibits "tricks," such as the one just mentioned, that may be dependent on the way in which Forth code is generated.)

Table Three. Code generated for six words with Voffs incorporated.

```
dup  (state = 0) ..... MOV   EAX, [EDI]+Voffs..... (new state = 1)
dup  (state = 1) ..... MOV   EBX, EAX..... (new state = 4)
dup  (state = 12) ..... <Voffs += -4>
                        MOV   [EDI]+Voffs, EBX
                        MOV   EDX, EBX..... (new state = 13)
drop (state = 0)..... <Voffs += +4> .....(state unchanged)
 ..... <no code generated>
drop (state = 1)..... <no code generated>..... (new state = 0)
drop (state = 12)..... <no code generated>..... (new state = 5)
swap (state = 0) ..... MOV   EAX, [EDI]+Voffs
                        MOV   EBX, [EDI]+Voffs+4
                        <Voffs += +8> ..... (new state = 6)
swap (state = 1) ..... MOV   EBX, [EDI]+Voffs
                        <Voffs += +4> ..... (new state = 6)
swap (state = 12) ..... <no code generated>..... (new state = 10)
rot  (state = 0) ..... MOV   EAX, [EDI]+Voffs
                        MOV   EBX, [EDI]+Voffs+4
                        MOV   EDX, [EDI]+Voffs+8
                        <Voffs += +12>..... (new state = 14)
rot  (state = 1) ..... MOV   EBX, [EDI]+Voffs
                        MOV   EDX, [EDI]+Voffs+4
                        <Voffs += +8> ..... (new state = 14)
rot  (state = 12) ..... <no code generated>..... (new state = 15)
xor  (state = 0) ..... MOV   EAX, [EDI]+Voffs
                        XOR   EAX, [EDI]+Voffs+4
                        <Voffs += +8> ..... (new state = 1)
xor  (state = 1) ..... XOR   EAX, [EDI]+Voffs
                        <Voffs += +4> .....(state unchanged)
xor  (state = 12) ..... XOR   EAX, EBX..... (new state = 5)
exit (state = 0) ..... MOV   EAX, [EDI]+Voffs
                        ADD   EDI, Voffs+4
                        RET
                        <Voffs = 0> ..... (new state = 1)
exit (state = 1) ..... ADD   EDI, Voffs
                        RET
                        <Voffs = 0> .....(state unchanged)
exit (state = 12) ..... ADD   EDI, Voffs-8
                        MOV   [EDI], EAX
                        MOV   [EDI]+4, EDX
                        MOV   EAX, EBX
                        RET
                        <Voffs = 0> ..... (new state = 1)
```

Figure Two-d. `xor`'s generated code and change-of-stack-state.



V. Combining Generated Code Instructions

The optimizations discussed previously relate to improving the speed of the generated code by reducing the number of instructions, and the number of memory accesses, needed to manipulate items on the data stack. There is an additional class of optimizations for the generated machine code, which takes advantage of the fact that generated code instructions can often be combined into a smaller number of faster instructions.

The most obvious area in which generated instructions can be combined is when literal values are used. Frequently, the literal is consumed by the next word. If the word following the literal is a simple word (i.e., a word that is subject to machine-code optimization), it is often possible to combine the literal with the code that is generated by the simple word. For example, the non-combined way of generating code for the sequence `15 +` from stack state 1 would be:[5]

```
MOV   EBX, 15
 (push literal 15 onto data stack)
ADD   EAX, EBX
(add pushed value to prior top item)
```

However, when these two code instructions are combined, a single code instruction can be used instead:

```
ADD   EAX, 15
(add literal 15 directly to top item)
```

When a named constant is called, it operates much like a literal, in that the constant value is pushed onto the data stack. A variable operates in much the same way, except that the constant being pushed is the variable's address. In both cases, the pushed value can often be combined into the next instruction in much the same way as a literal. For example, the code instructions to access the contents of a variable, as generated by the sequence `v1 @` (where `v1` is defined by the phrase `variable v1`), might initially appear as:

[5] In all the examples in this section, I am assuming that the starting stack state is state 1; the use of Voffs is omitted.

MOV EBX, offset v1[6]
MOV EBX, [EBX]

These instructions can be replaced by the single code instruction:
MOV EBX, v1

Another common situation occurs where a variable's contents are accessed, then consumed by the following word. For example, the sequence `v1 @ +` might generate the following code without combining:
MOV EBX, offset v1
MOV EBX, [EBX]
ADD EAX, EBX

When combining is used, the above three-machine-code instructions are replaced with a single instruction:
ADD EAX, v1

Other combinations are possible as well. For example, an index may be scaled, then added to the base address of an array of 32-bit values; the indexed element is then accessed. An example is the sequence `array1 v1 @ 4* + @` (where `array1` returns the starting address of an array of 32-bit values). The resulting generated code could take advantage of the indexed addressing mode to allow otherwise-separate instructions to be combined:
MOV EBX, v1
MOV EBX, array1[4*EBX]

It also makes sense to perform computations involving literals during code generation. For example, the word sequence `c1 4* 1+ +` (where `c1` is defined by the phrase `30 constant c1`) may generate the code instruction:
ADD 121 (`c1` = 30, so 4 * `c1` + 1 = 121)

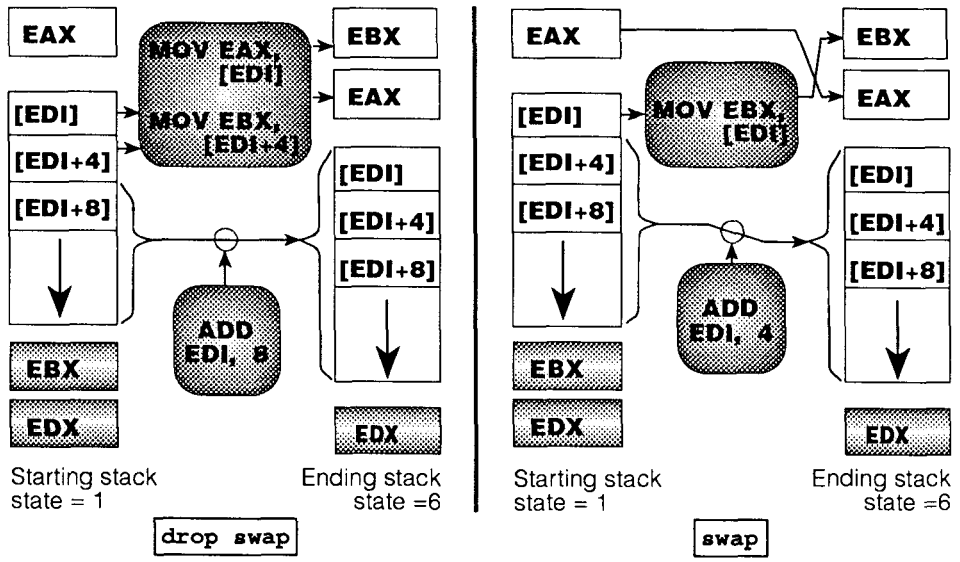
This also applies to address calculations, such as for the sequence `array1 20 + @`:
MOV EBX, array1+20

Another situation where combining code instructions is particularly useful occurs when a test is followed immediately by a control-structure word such as `if` or `until`. The need to generate an explicit flag is thus eliminated, and the test plus control-structure branch can often be reduced to from two to four instructions. For example, the sequence `= if ... then` might generate the following code:
MOV EBX, [EDI]
ADD EDI, 4
CMP EAX, EBX
JNE _LBL_nnn
(_LBL_nnn is a compiler-generated jump label)
...
_LBL_nnn:

Of course, a comparison with a literal or constant value generates even simpler code. For example, the sequence `8 < if ... then` might generate the following code:
CMP EAX, 8
JNL _LBL_nnn
...

[6] For those of you not familiar with Intel assembler notation, the use of *offset* before an address value causes the address value to be treated as a literal. If an address value is *not* preceded by *offset*, the corresponding memory location is accessed instead. However, a non-address value does not need *offset* to be treated as a literal.

Figure Three. Code generated by `swap` with and without preceding `drop`.



(`drop` generates no code itself, but it changes the stack state to 0 before `swap` generates *its* code!)

Figure Four. Illustration of the virtual offset.

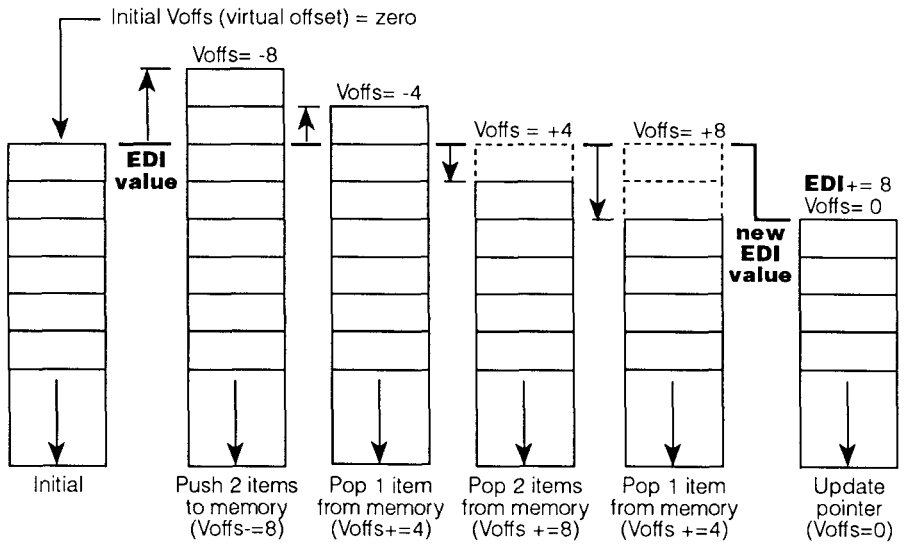
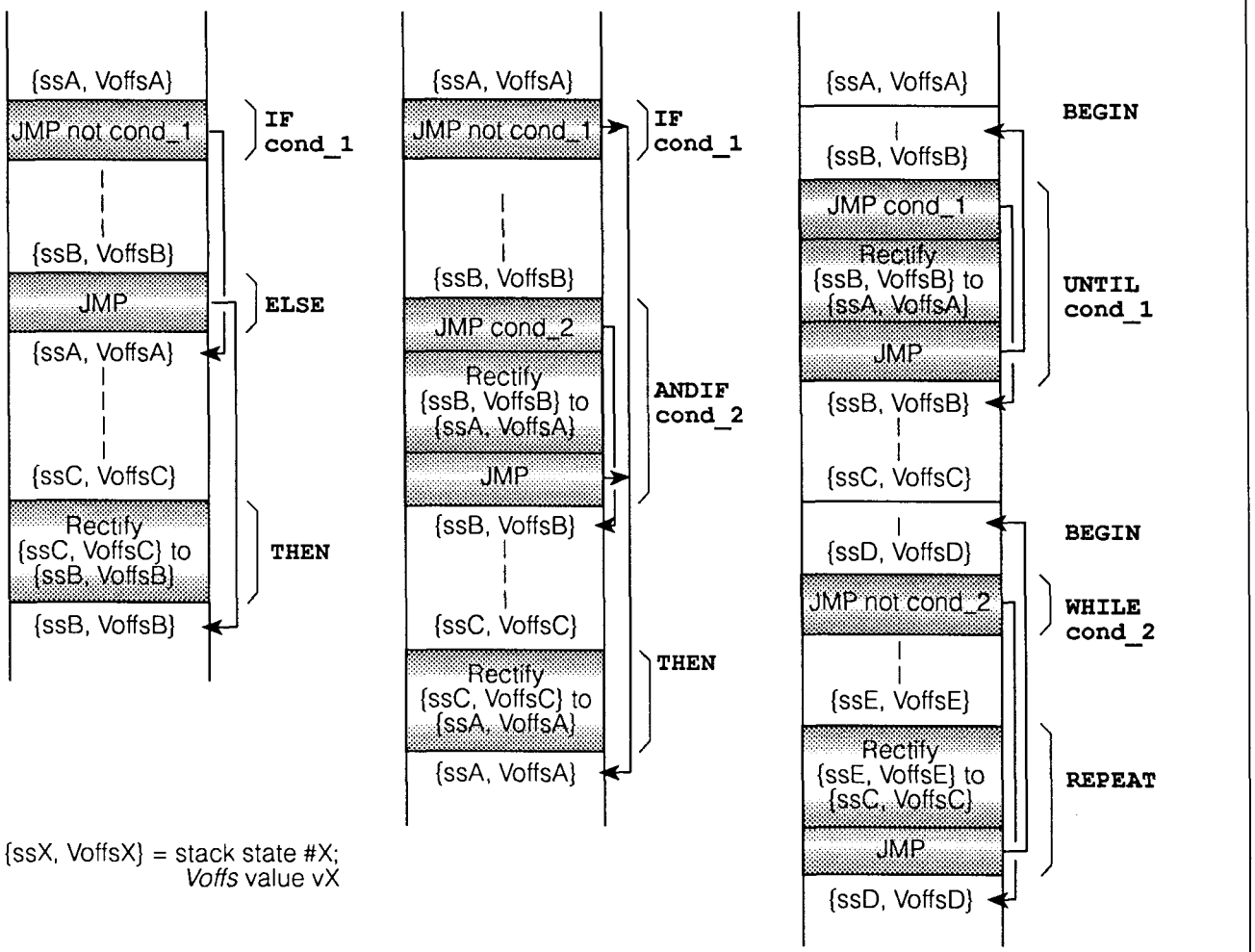


Figure Five. Examples of rectifying the stack state and Voffs value for various control-structure jumps.



_LBL_nnn:

To implement the combining of machine instructions, the compiler looks for sequences of simple Forth words that could potentially generate combinable instructions. In practice, an array is used to retain up to some maximum number of entries that contain tokens for words that have been recently parsed off during compilation, but for which no machine code has yet been generated. When the array is full, a pattern-matching scan is started on the array, beginning with the first entry. A set of code-generating patterns is matched, one pattern at a time, against the array's contents. Longer patterns are scanned before shorter ones, but the matched pattern must always begin with the first entry in the array. Since patterns are defined for all possible single tokens, at least one token in the array is guaranteed to be matched.

When a matching pattern is found in the array, the matched entries are removed from the array, and the remaining entries are moved down to fill in the vacated entries.[7] (If computations involving literals or constants are involved, the removed entries may instead be replaced by a smaller number of entries.) The matched pattern is associated with the code to be generated for that pattern. Since at least one entry will now be vacant at the end of the array, the compiler can then continue to parse off at least one additional word. Note that pattern-matching on the array is performed only when it is full, or when the compiler must force the entire array to be compiled into code and then emptied. The latter situation may occur when a non-simple word (such as another compiled word) is called from the currently compiled word, when an `exit` is encountered, or when a branch target is reached. Figure Six illustrates an example of the pattern-matching process for a simple (if contrived) setup.[8]

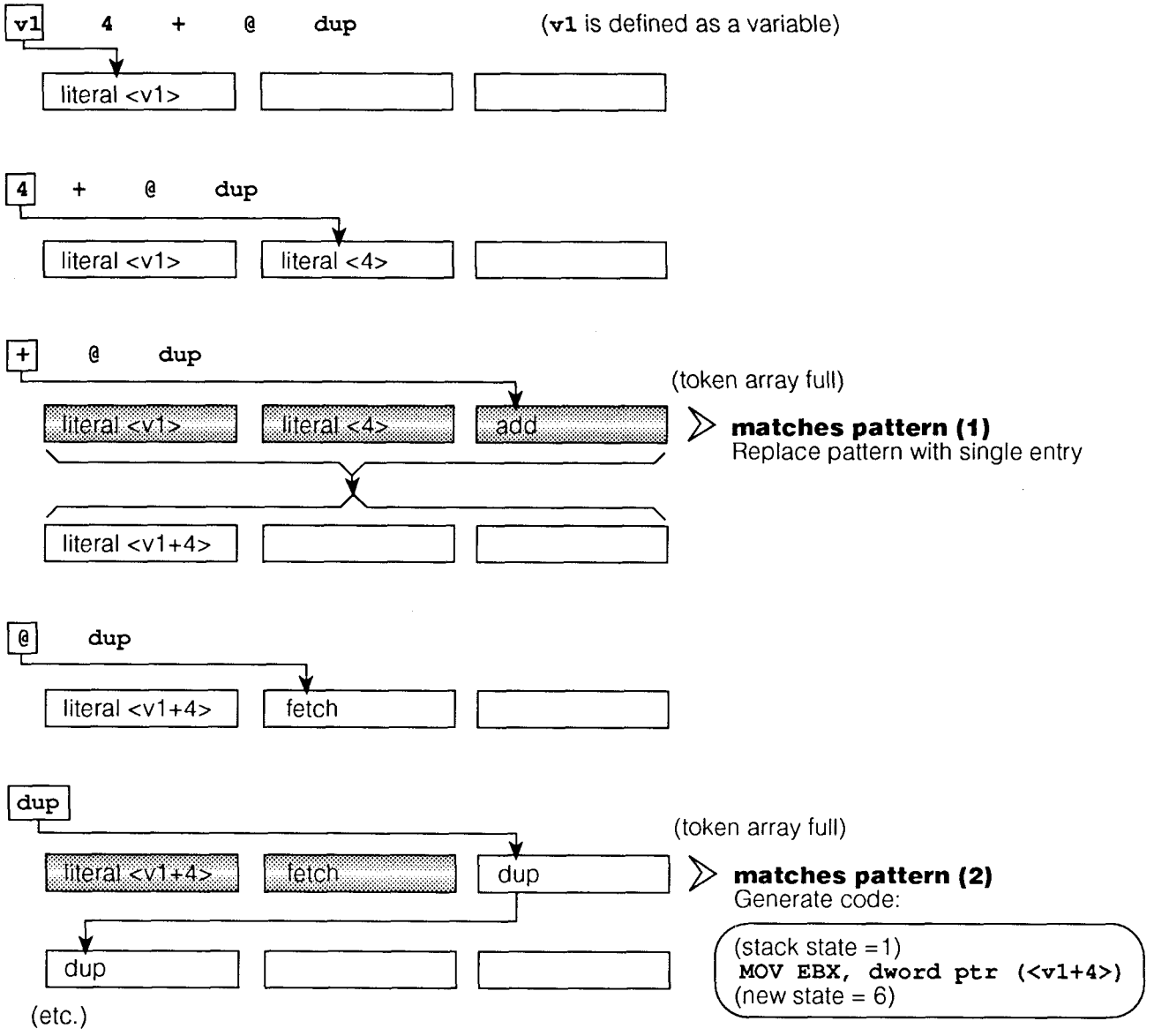
[7] In order to avoid the overhead of actually moving array entries down after earlier entries have been removed, the array should, in practice, be implemented as a circular queue. The length of the array should be at least equal to the length of the longest code-generating pattern. However, the use of "tokens" permits the individual entries in both the array and the patterns to be reduced down to one or two bytes apiece; it also speeds up the process of matching patterns to the array's contents.

In the case of literal or constant values (including the starting addresses of arrays), the associated values may be kept in a separate, parallel array, where each entry is 32 bits (four bytes). For certain commonly used constants, such as 0, 1, 2, and -1, special "tokens" may be used that permit generation of code that can be optimized for the presence of such constants.

[8] There are two additional areas where optimizations may have a significant impact. The first area is in replacing multiplications by "small" constants with sequences of shifts, additions, and/or subtractions. The second area involves the multiplication of a loop index by a constant value; the multiplication of the index can be replaced by maintenance of a "shadow" index in parallel with the "regular" loop index. (Multiple "shadow" indexes may be associated with a single "regular" index.)

Both of these areas of optimization are beyond the scope of this article. Interested readers are referred to the sections on code optimization in the book *Principles of Compiler Design* by Aho and Ullman, published by Addison-Wesley. (Computer science students may recognize this book as the infamous "dragon book" because of the cover design.)

Figure Six. Illustration of the pattern-matching process for a simple set of patterns.



Set of matching token patterns (partial):

(1) literal <n1>; literal <n2>; add	Replace pattern with single entry: literal <n1+n2>			
(2) literal <n>; fetch	(stack state = 1)	`MOV EBX, dword ptr (<n>)`		(new state = 6)
(3) literal <n>	(stack state = 1)	`MOV EBX, offset (<n>)`		(new state = 6)
(4) add	(stack state = 1)	`ADD EAX, [EDI]+Voffs`		
		`Voffs += +4`		(state unchanged)
(5) dup	(stack state = 6)	`MOV EDX, EBX`		(new state = 13)
(6) fetch	(stack state = 6)	`MOV EBX, [EBX]`		(state unchanged)

VI. Macros and Macro Compilation

One major problem with the foregoing optimization techniques is that, in many typical compiled Forth words, simple words may be thoroughly intermixed with calls to other compiled words. Unfortunately, calls to compiled words can severely limit the opportunities for code optimization, because of their nature:

1. A monkey wrench gets thrown into optimization of operations on the data stack by calls to compiled words, because each call to a compiled word requires that both the stack state and the value of Voffs be rectified to values consistent with the expectations already compiled into each and every compiled word. By contrast, when sequences of source code contain only simple words, the compiler has considerably more flexibility in how it generates its code with respect to the optimization of data-stack handling.
2. In addition, there is simply no way to combine a call to a compiled word with any other machine instruction. Thus, opportunities for code optimization due to the

combining of instructions are also severely limited.

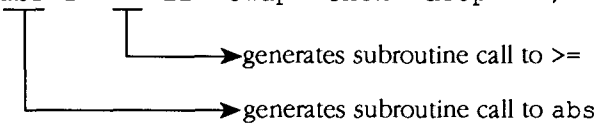
What is needed is a mechanism whereby calls to compiled words can be replaced by in-line code sequences. An in-line code sequence is simply the same sequence of words that would otherwise be present in the compiled word. (Of course, the terminating `exit` would be omitted, and several other words may not be equivalent when used in-line vs. in a compiled word's definition.)

For example, the words `abs` and `>=` have the following Forth source definitions:

```
: abs  dup  0<  if  negate  then    ;
: >=   <  not                       ;
```

If these words are later compiled into another word, the generated code would make subroutine calls to them, as in the following example:

```
: deadzone
  ( n1 n2 n3  --  n1  if |n1-n2| < n3 )
  ( n1 n2 n3  --  n2  if |n1-n2| >= n3 )
  >r  over  over  -
  abs r> >= if  swap  then  drop    ;
```



However, if the words `abs` and `>=` could be set up as in-line routines, the above example for `deadzone` would generate code as if the *source* definitions for `abs` and `>=` were substituted at the appropriate places:

```
: deadzone
  >r  over  over  -
  dup  0<  if  negate  then
  ( abs definition substituted)
  r>
  <  not
  ( >= definition substituted)
  if  swap  then  drop  ;
```

Needless to say, there are more opportunities for code optimization in the second definition of `deadzone`, especially with regard to execution speed. However, since the compiled words are expressed in their full, in-line form in the second definition, the resulting code may be somewhat larger. In the typical 386 environment, program size is less of an issue than it is in some other environments, while speed continues to be a significant concern, especially in such compute-intensive applications as graphical user interfaces, animation, and multimedia.

I propose that the following mechanism be implemented in the compiler to permit "macros" (i.e., in-line code definitions) to be created: namely that such definitions use the word `:macro` to start their definitions, rather than the usual : (colon) word. In effect, every word within the macro definition, except for the terminating ; (semicolon) word, is deferred, rather than compiled in the usual fashion. If the definition is followed by the word `executable`, an interpret-time alias is created for the macro; the alias, which has the same name as the macro, is then executable in interpretive mode.[9] The following gives modified definitions for `abs` and `>=` using macro definitions:

```
:macro  abs
        dup 0< if negate then  ;
        executable

:macro  >=
        < not  ; executable
```

An additional word, `does>macro`, completes the basic macro facility. The word `does>macro` bears the same relationship to `does>` that `:macro` bears to the : (colon) word.

The macro facility just described can be extended in a number of ways. The principal way in which it could be extended would be to introduce words that are *not* deferred within a macro definition. This opens up some new possibilities, such as conditional compilation via macros. However, my primary purpose in introducing the macro capability in this article, was to show how it could be used to permit better code optimization via the in-line compiling capability.

VII. Conclusion

A Forth implementation may compile words in such a way as to generate machine-language code, rather than just pointers that require additional run-time interpretation. Such machine code can be optimized so that it runs even faster. There is sometimes a savings in space as well, although space savings may not be as critical an issue, especially for typical 386-based PC's.

While I used the Intel 386 to illustrate my examples, the basic ideas can be transferred to other processors as well. This is especially true of the Motorola 68000 family. The optimizations I have presented are particularly oriented toward the generation of optimized code for a Forth compiler, although some of the techniques can be applied to other languages as well. Likewise, some (though not all) optimization techniques used with other compilers may be profitably applied to a Forth compiler.

In order to take maximum advantage of optimization possibilities, the Forth compiler should be extended with a facility for in-line substitution of source code definitions, in lieu of the compiling of subroutine calls that impose severe restrictions on code optimization. I have presented the definition of a basic macro facility that fills this purpose and opens up some new possibilities in its own right.

[9] Since all words are *deferred* in a macro, it is entirely possible to have a macro that has incomplete control structures. For example, a definition such as the following is perfectly legitimate as a macro:

```
:macro  notif   not if  ;
```

However, such a macro should *not* be made executable at interpret time, because of the incomplete `if` control structure (`then` is missing). If you use `executable` on this macro, you will get a compiler error message.

The author, a consultant specializing in PC-based Forth, has more than six years of experience with Forth and related systems; the company he owns is engaged in development of a major Forth for Microsoft Windows 3.1. If you have questions or comments on this article, address them to David M. Sanders, PSIQ FOUR Enterprises, 370 Turk St., Suite #123, San Francisco, California 94102.

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

Forth: Always New, Despite its Age

Whenever we send out a Forth marketing message, we need to speak of Forth using the latest terminology on the high-tech horizon. Two terms to consider with regard to Forth are "application framework" and "introspection."

Did you know that, among other things, Forth is a *development system framework* and a *programming language framework?* As the term framework implies, it frames your program with a structure to carry the code you supply.

A framework offers another form of code reuse. The most familiar format for reusable code is a library, which consists of bundled service-providing routines that you choose to call or not. Besides offering already-made services, the framework "prewires" certain services together to roughly model the application you are about to build. The framework offers a basic design for accomplishing a task, instead of offering piecemeal code that you may use along with your own design.

Frameworks such as HyperCard and MacApp are helping application programmers create GUI applications for the Macintosh with greatly reduced effort. For PCs running Windows, the premier framework is Visual BASIC.

The actual processor is less of a concern than the virtual processor...

The code from the framework satisfies general goals, such as queuing the input events associated with a mouse and keyboard. Application-specific goals are met by your own routines, which the framework has a way to call at the appropriate times.

Forth can be seen not merely as an application framework (particularly if your final application preserves the Forth interpreter), but also as a development system framework and a programming language framework.

The Forth programming language is subject to modification by the programmer. Typically, control structures are written to replace the ones that came with a system. Because it does not involve redesigning Forth from the ground up, this type of language refinement is equivalent to the use of a framework. In this way, Forth functions as a programming language framework. (The end product is a Forth dialect or a new language—typically one that preserves the basic Forth compiler.)

Habitually, Forth programmers create decompilers, tracers, code profilers, and other tools. In Forth, they are easy to create. In this case, Forth is functioning as a development system framework—allowing you to create specific development tools within an overall tool framework. (The end product is a customized application development system that preserves most of Forth's native development tools.)

The term *framework* helps convey an important message to the larger programming community. It is one way that Forth can be considered a very progressive, and a very high-level, environment.

Programs Become Introspective

Another new term we can use to describe Forth involves the concept of introspection. This term came to my attention recently with regard to Dylan. The Dylan programming language is a product of long-term research at Apple Computer. Dylan is object oriented, so the concept of introspection is the concept of self-identifying objects:

> The [Dylan] language should contain features for introspection. This means that the language run-time should have sufficient power to answer questions about itself and the objects it manages. For example, it should be possible at execution time to analyze the structure of an object, find the subclasses of a class, etc.
>
> To facilitate type-safety and introspection, objects are self-identifying in memory. Unless all uses of an object can be analyzed at compile-time, the run-time memory for the object should contain enough information to identify its class and value. —*Dylan Language Reference*

The ease with which Forth decompilers can be written suggests that Forth compilers already produce memory images that are subject to meaningful inspection.

Several Forth programmers at the Asilomar FORML conference last year were interested in developing Forth

systems that relied on decompilers to allow browsing and editing of compiled routines in terms of their computed source code.

At the Silicon Valley FIG Chapter meetings, there has been talk of building decompilers to display the source code corresponding to the compiled code in several languages, including Forth and C. That kind of translation could eventually provide cross-platform tools for working simultaneously in Forth and C environments.

(Another approach to mixed language support might be to implement C in such a way that its run-time system supports a two-stack architecture. Currently, C uses one stack to support function returns as well as the parameterization of routines. It does not sound as though it would be a difficult for C to accept a two-stack architecture. The challenge would be to make sure that the switchover was seamlessly transparent to C programmers.)

Why Get Real, When Virtual Reality is Better?

Forth is for programmers who want to be assured that the code that runs on the processor is their own creation, not the compiler's creation. Insofar as high-level Forth instructions are ultimately mapped into the native instructions of a given processor, the preceding statement is slightly misleading. However, due to Forth's lack of syntax and lack of intense processing as part of its compiling process, the statement still rings essentially true.

Yes, the Forth implementor has written most of the low-level code that ultimately runs on the processor. But the actual processor is less of a concern than the virtual processor—control of which is the object of desire for the Forth programmer. We are much more able to realize our dreams when the medium for their creation is virtual reality.

Forth gives the illusion of having assembly-language levels of control to do whatever needs doing, but without the usual inflexibility of an assembly language development system.

Forth is a carefully chosen *virtual processor* language that is capable of being very sweeping, as well as being very granular. It can be made as high level as desired because the virtual Forth processor's instruction set is extensible in a multitude of ways. We can create compilers for new data structures and compiler extensions that make sophisticated routines easier to specify. We can even create new compiling systems based on the compiler framework supplied with Forth. Metacompilers and target Forth compilers are two examples of added-on systems often developed using Forth's compiler-compiler tools.

Perhaps the virtual component of our pursuit of Forth is the central one. Perhaps, because we are unable or unwilling to deal with reality, we make up our own reality to suit our needs. Forth offers us a virtual programming environment (or programming language framework) whose content is placed under our control. After our flights of fancy without real-world constraints, we become hooked without hope of recovery.

Someday, when Forth is regularly implemented in silicon, we'll get back to nonvirtual reality.

—Mike Elola

Product Watch

NOVEMBER 1993

FORTH, Inc. announced a new cross-compiler to support the Motorola 68HC16 microcontroller, which includes Background Debugging Mode (BDM). With BDM, thorough software tests can be performed without in-circuit emulators or other special hardware.

The new software is supplied along with the Motorola M68HC16Z1EVB development board. It provides 64K bytes of EPROM or RAM (expandable to 128K), a Centronics-compatible parallel port used by the BDM, and one RS-232 port. The new chipForth is tailored to work with this target. It can also be configured to support other boards containing the 68HC16 target processor.

From a host PC, chipForth provides interactive, incremental programming. The host PC runs the polyForth development environment under MS-DOS. The target runs a real-time multi-tasking executive and a kernel including many useful primitives. Source code is provided for the target software, the cross-assembler, and the cross-compiler.

The 68HC16 version of chipForth is the latest addition to FORTH, Inc.'s line of development systems suited to real-time and device control applications. Other versions support the 8031/51, 80x96, 68HC11, and 68xxx families of microcontrollers.

DECEMBER 1993

Creative Solutions, Inc. announced that it will provide a version of MacForth to run on Apple's new line of RISC computers based on the Power PC microprocessor. Apple expects to ship its new class of computers in the first half of 1994.

CSI also announced the Hustler HDS+ series of plug-in boards for Macintosh computers. The first of this series is a new board with high-speed serial ports (230,000 baud on one port or 115,000 baud on two ports) and a 64K data buffer. It had an introductory price of $299 when it was announced.

COMPANIES MENTIONED

Forth Inc.
111 N. Sepulveda Blvd.
Manhattan Beach, California 90266-6847
Fax: 310-318-7130
Phone: 310-372-8493

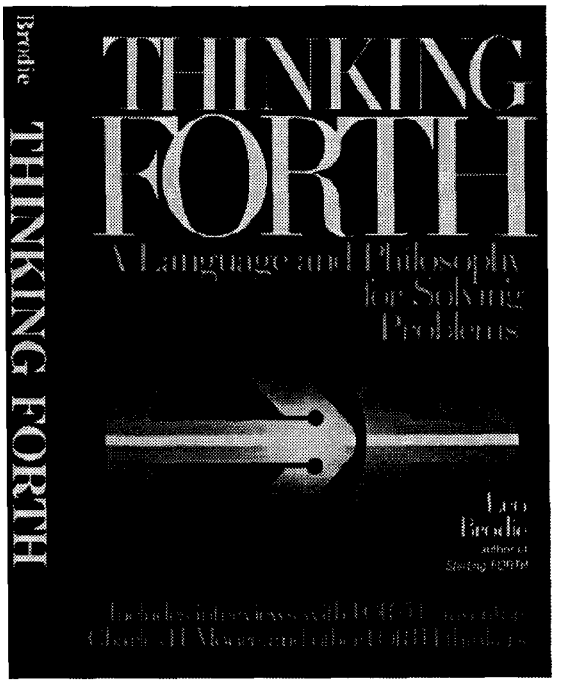
Creative Solutions, Inc.
4701 Randolph Road, Suite 12
Rockville, Maryland 20852
Phone: 301-984-0262

First in a new series of books by the Forth Interest Group

Business, industry, and education are discovering that FORTH is an especially effective language for producing compact, efficient applications for realtime, real-world tasks. And now there's Thinking Forth—an instructive guide that illustrates the *elegant logic* behind the language and shows how to apply specific problem-solving tools to software, regardless of your programming environment.

It combines the philosophy behind Forth with traditional, disciplined approaches to software development—to give you a basis for writing more readable, easier-to-write, and easier-to-maintain software applications in any language.

Written in the same lucid, humorous style as the author's *Starting Forth* and packed with detailed coding samples aand illustrations, Thinking Forth reviews fundamental Forth concepts and takes you from the initial specification of your software project through the analysis and implementation process, showing how to simplify your program and still keep it flexible throughout. Both beginning and experienced programmers will gain a better understanding and mastery of such topics as



Cover design for illustration design will differ

- FORTH style and conventions
- decomposition
- factoring
- handling data
- simplifying control structures
- and more.

And, to give you an idea of how these concepts can be applied, Thinking Forth contains revealing interviews with real-life users and with Forth's creator, Charles H. Moore.

To program intelligently, you must first *think* intelligently, and that's where Thinking Forth comes in.

Available in April

$20