IT'S ALL HERE:
SPEED, TOOLS, FLEXIBILITY
THE PC-RISC SYSTEM

■ SPEED
  • NC4016 RISC Engine
  • 4 MHz operation runs at 5 MIPS
  • 5 MHz operation runs at 6 MIPS
  • 6 MHz operation runs at 7 MIPS
  • Up to 40 MIPS with multiple PC4000s

■ TOOLS
  SCForth2
  • Multitasker switches tasks in 24 cycles
  • Compiler optimizes over 150 phrases
  • Loads from block or ASCII text files

  PCX
  • PC/PC4000 interface software with windowing
  • On line help screens and full featured editor
  • Concurrent PC operation

  SC-C
  • K&R standard implementation for NC4016
  • Full 64K word address space
  • In-line assembly code

  SCMacro
  • Full featured native code assembler
  • Includes linker, editor and image generator

■ FLEXIBILITY
  • Coprocessor board for PC, XT, or AT
  • General purpose development system
  • Ideal for control or image processing
  • OEM product inclusion

  PC4000 4MHz .... $1,295
  PC4000 5MHz .... $1,495
  PC4000 6MHz .... $1,695
  SCForth2 ......... $595
  SC-C ............. $595
  SCMacro ........... $595

SILICON COMPOSERS, 210 California Avenue, Palo Alto, CA 94306 (415) 322-8763
Anonymous variables aren’t the only way to implement local variables. An alternative is to copy or move items from the stack to storage allocated in the word that uses them. We can achieve this and preserve a convenient and readable syntax.

Forth definitions typically deal with memory locations within the dictionary boundaries. But special problems arise with PROM-based systems. Usually, a target system will have a separate area of RAM for the storing variables. The task is to evolve a system of vectoring variable operations into this RAM area.

These entrepreneurs went from college courses to professional programming, followed quickly by designing, writing, and selling Mach 2, their Forth for 68000-based micros. Michael Ham continues his series of interviews with Lori Chavez and Derrick Miley, advocates of an integrated, interactive Forth environment.

The author of MVP-FORTH Integer and Floating Point Math had to implement quick, accurate, and relatively compact math functions. His research resulted in the equations presented here. (Don’t even ask about the derivations....)
EDITORIAL

Time was, computer-users groups were the loci of the micro revolutionaries, or at least of those who iconized them. They were like open-seating concerts for LED-heads who derived their main satisfaction from affiliation itself. At the meetings, whiz kids and garage-shop pioneers found other seekers of the satisfaction that accrues from each computing challenge well met. They ran improvisational meetings, or else surrendered to the procedural overtures and organizing committees of FIG chapters; and groups need more from their meetings than a visitation of the copyright holder. Changes to the fall season brings two of the Forth Interest Group’s major events. Many of you will see this issue first at the Forth National Convention or the following FORML conference. We will cover highlights of those events in upcoming issues for those who cannot attend.

Meanwhile, keep working on your letters and articles for Forth Dimensions. We intend to publish the best and most interesting work from the Forth community, which is only possible if every reader thinks of himself as a potential contributing author. Write to the FIG office for a copy of the latest writers guidelines, we’d like to hear from you!

—Marlin Ouwerson
Editor

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

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

Copyright © 1987 by Forth Interest Group, Inc. The material contained in this periodical (but not the code) is copyrighted by the individual authors of the articles and by Forth Interest Group, Inc., respectively. Any reproduction or use of this periodical as it is compiled or the articles, except reproductions for non-commercial purposes, without the written permission of Forth Interest Group, Inc. is a violation of the Copyright Laws. Any code bearing a copyright notice, however, can be used only with permission of the copyright holder.

About the Forth Interest Group

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

“Forth Dimensions” is published bi-monthly for $24/36 per year by the Forth Interest Group, 1330 S. Bascom Ave., Suite D, San Jose, CA 95128. Second-class postage pending at San Jose, CA 95101. POSTMASTER: Send address changes to Forth Dimensions, P.O. Box 8231, San Jose, CA 95155.”

Forth Dimensions
Published by the
Forth Interest Group
Volume IX, Number 4
November/December 1987
Editor
Marlin Ouwerson
Advertising Manager
Kent Safford
Design and Production
Berglund Graphics
ISSN#0884-0822

Forth Dimensions
Volume IX, Number 4
Novix Finbonacci

Dear Mr. Ouverson:

The July and August 1987 editions of BYTE compare various personal computers running six benchmarks. The August edition contains a comparison of the times given for their first benchmark, an algorithm that computes the 24th Fibonacci number 100 times. The times I have achieved on two systems using the Novix NC4016 Forth engine are:

<table>
<thead>
<tr>
<th>System</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delta Board:</td>
<td>28.2</td>
</tr>
<tr>
<td>(4 Mhz NC4016 CPU)</td>
<td></td>
</tr>
<tr>
<td>PC4000:</td>
<td>22.6</td>
</tr>
<tr>
<td>(5 Mhz NC4016 CPU)</td>
<td></td>
</tr>
</tbody>
</table>

As can be seen by comparing these times to those in the article, the NC4016—running at a clock rate four times as slow as a DESKPRO 386—executes this benchmark nearly twice as fast. Also, the cmFORTH implementation is much more concise and (to me, at least) less cryptic than the C version. And, of course, the word FIB can be run interactively and used independently of FIBTEST.

With this benchmark, the power of Forth and the NC4016 is directly shown. Just imagine what can be done with an NC4016 running at 16 Mhz!

Douglas Ross
NASA
Goddard Space Flight Center
Greenbelt, Maryland 20771

New SEARCH for (Z80) F83

Dear Mr. Ouverson,

R.L. Hoffpauer’s code SEARCH (FD IX/2) was a very welcome addition to my already cluttered desk. I had been trying, off and on, to write machine code to replace the high-level F83 SEARCH on my Z80-based system, and was getting nowhere fast. Not being an 8088 programmer, I hadn’t thought of using an index, not only to point, but also to count. This is what comes of not working on IBM PCs.

But while I was doing the conversion, I realized that Mr. Hoffpauer’s code can be improved, both from a theoretical and a practical standpoint.

Theory first: Mr. Hoffpauer uses an UNTIL structure for his main loop. This is erroneous, since it means the routine will go right through a loop even if the string sought is longer than the buffer. What is actually needed is a WHILE (because we may not do any comparing at all). On the other hand, Mr. Hoffpauer uses a WHILE for his inner loop, i.e., comparison of the string once the first character has been found. This is again mistaken: an UNTIL structure really is necessary this time because we will, in any event, make at least one comparison. All we have to do is initialize the inner-loop string pointer/counter to zero, then increment it at the start of the loop, rather than at the end. And, of course, change the test.

Forth is, after all, a structured language. Let’s keep it that way, even when we’re translating our algorithms into machine code; and it’s essential that algorithms use the correct structures for a given problem.

On the practical side, I decided to shorten the code by storing the initial character of the string in an anonymous variable. This way, it is only tested for capitalization once, before the main loop begins, and thereafter is read directly in the form needed for comparison. (This gives rise to some stack gymnastics at assembly time.) I also took advantage of the fact that F83 uses pure flags to test by INCing BX instead of CMPing it with zero, saving space and time on each test.

The accompanying screens give my Z80 version (in 8080, because that was the assembler incorporated in the F83 I bought; and I cannot metacompile a version with a Z80 assembler, because my
ASSEMBLER LABEL >UP
ASCII a # AL CMP >=
IF ASCII z # AL CMP <= IF 20 # AL SUB THEN THEN RET

HERE ( Hold address) NOP \ Reserve one byte for string initial

CODE SEARCH (s sadr slen badr blen -- offset flag)
CLUD
CX POP BX POP \ Direction
DX POP DI POP \ CX <- blen BX <- badr
DX CX SUB BX CX ADD \ DX <- slen DI <- sadr
SI PUSH BX PUSH \ CX <- last address
BX SI XCHG \ Save IP and badr
0 [DI] AL MOV \ SI <- badr
CAPS #) BX MOV BX INC 0=
IF >UP #) CALL THEN \ 1st char of string
AL OVER ( Assemble address) #) MOV \ Case sensitive?
BEGIN
CX SI CMP <= \ Convert if not
WHILE
4 ROLL ( Assemble adr) #) AH MOV \ Store it
AL LODS
CAPS #) BX MOV BX INC 0=
IF >UP #) CALL THEN
AH AL CMP 0=
IF
SI PUSH
0 # BX MOV \ Loop WHILE in buffer
BEGIN
BX INC
0 [DI+BX] AL MOV
BX PUSH
CAPS #) BX MOV BX INC 0=
IF >UP #) CALL THEN
AH AL XCHG
AL LODS
CAPS #) BX MOV BX INC 0=
IF >UP #) CALL THEN
BX POP
AH AL CMP 0<> \ Get str-initial to AH
UNTIL
SI POP
DX BX CMP >= \ Get char from buffer
IF
BX POP
BX SI SUB SI DEC
DX POP
DX SI XCHG
-1 # AX MOV
2PUSH
THEN
THEN
REPEAT
DX POP
SI POP \ Clean up stack
AX AX XOR
2PUSH
END-CODE

\ Get Forth IP
\ Send false flag
Computer only has about 12K workspace — see note on IBMs above). I also propose my modification of Mr. Hoffpauer's code. Thanks a lot to the whole team for the magazine.

Sincerely,
Martin Guy
9 rue de la Peupleraie
71500 Chateaurenard
France

Batcher's Last Re-Sort
Dear Marlin,

I was very gratified to see the responses to the article about Batcher's Sort. I would like the chance to comment on some of the things that were not clear in the article.

Mr. Anway is correct. The flag-passing involving Q is not as clear as it could be. Originally, Q existed only as a stack value, and was made explicit only for publication. Doing this as an afterthought resulted in a messy structure, which was cleared up thanks to Mr. Anway's help.

Mr. Thomas' letter initially shocked me. It is a fundamental characteristic of Batcher's method that it is data independent, so I was really surprised by Mr. Thomas' claim to the contrary. I double-checked Knuth and verified that duplicate data values are explicitly allowed. Then I tried sorting data with duplicates, as Mr. Thomas suggested. No difficulty: He reads the word 2**N as the square of an argument N, whereas the intended meaning is to return 2 raised to the Nth power. Mr. Anway's code presents a correct implementation of 2**N. I had thought this was in the Forth-79 Standard Reference Word Set, so I didn't define it in the article. On checking, I find that it is not there, and I apologize for this omission and the confusion it may have caused.

I have since found an unusual use of sorting I'd like to share: I have an application that acquires a signal from a fast ADC and plots it in graphics. Useful insights into the experiment being performed can be obtained by sorting the data and reploting it. The sorted data is often grouped around several fixed levels, rather than being smoothly distributed, as I had expected — this was not obvious from looking at plots of the original data.

Sincerely yours,
John Konopka
c/o Kevex Corporation
1101 Chess Drive
Foster City, California 94404

Tim Lee's Long Names
Dear Marlin,

I was pleased to receive the courtesy copies of Forth Dimensions containing Mike Ham's interview with me. Thank you!

While reading the interview, I had an impression similar to hearing my own voice on tape ("Is that what I sound like?"). One thing in particular made me laugh, the part that has me endorsing the use of long definitions without comments!

Well, I'd like to set the record straight on this: what I meant to say is that I'm using longer word names and shorter definitions. When I write a new definition, I try to make the words it contains read in English as well as in Forth. For example:

```forth
: dither_screen ( -- )
dither_color
graphics_page_fill ;
```

If it happens that, after combining definitions made this way, the resulting definition doesn't express a clear idea in English, it is often a clue that I haven't partitioned the problem correctly. Then I revise the names and/or functions of the words that compose the current definition.

I've been using this method for the past year, and continue to find it rewarding in the solutions that it reveals. The few extra characters required to specify a longer name each time I type it, is a cost that is offset by being able to work at a higher level of abstraction. (There are functions that cannot be accurately identified with a single word. The blurring of distinctions that result from combining fuzzily named functions limits the level of abstraction that can be attained.)

Please publish this clarification so impressionable new Forth programmers don't start writing lengthy, uncommented definitions; and so my more experienced friends won't point and laugh when I show up at the Forth Convention!

Sincerely,

Tim Lee
Binary Systems

(Continued on page 15.)
### Guy Screens:

<table>
<thead>
<tr>
<th>Ecran 8</th>
<th>Ecram 23</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 \ Code SEARCH</td>
<td>22aug87ajg \ Code SEARCH</td>
</tr>
<tr>
<td>1 \</td>
<td>23aug87ajg</td>
</tr>
<tr>
<td>1 \</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Defining IX here makes life easier in a few seconds.</td>
</tr>
<tr>
<td>3</td>
<td>N.B. 52ED, is for instruction 5BC, HL, DE</td>
</tr>
<tr>
<td>4</td>
<td>Lots of stack work to get the following allocations:</td>
</tr>
<tr>
<td>5</td>
<td>DE is buffer pointer, BC points at start of string to seek,</td>
</tr>
<tr>
<td>6</td>
<td>HL holds string length, IX address of last possible match.</td>
</tr>
<tr>
<td>7</td>
<td>Check if last address has been reached, and</td>
</tr>
<tr>
<td>8</td>
<td>while it hasn't, do the loop.</td>
</tr>
<tr>
<td>9</td>
<td>Get first char from string and compare with next one in buffer.</td>
</tr>
<tr>
<td>10</td>
<td>If the first character matches,</td>
</tr>
<tr>
<td>11</td>
<td>save b-ptr and s-len, and use HL as index into string.</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

---

### Code SEARCH

<table>
<thead>
<tr>
<th>Ecram 9</th>
<th>Ecram 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 \ Code SEARCH</td>
<td>23aug87ajg \ Code SEARCH</td>
</tr>
<tr>
<td>1 \</td>
<td>23aug87ajg</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Get next character from string, convert if necessary, and</td>
</tr>
<tr>
<td>4</td>
<td>compare with next character in buffer.</td>
</tr>
<tr>
<td>5</td>
<td>Loop on this until we find different characters.</td>
</tr>
<tr>
<td>6</td>
<td>Then subtract s-len from found-len, restore HL = s-len.</td>
</tr>
<tr>
<td>7</td>
<td>If found-len != s-len, we have a match.</td>
</tr>
<tr>
<td>8</td>
<td>So calculate the offset and store it in DE.</td>
</tr>
<tr>
<td>9</td>
<td>Then restore the Forth IP and send back offset and true flag.</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>End of main loop: we leave here if end of buffer and no match.</td>
</tr>
<tr>
<td>12</td>
<td>So get junk (b-adr) to DE, restore IP, send back false flag.</td>
</tr>
</tbody>
</table>

---

## Go FORTH™

**The ProDOS Forth Language implementation for the Apple Computer //e, //c, //gs and ///**

FORTH is more than just a high level language that combines many of the features of other computer languages. It is a development environment and a method of approaching problem solving. FORTH is a 'grass roots' language, developed and enhanced in the real world by working programmers who needed a language that they could USE. Many of the concepts of FORTH are several years ahead of other languages of today. It is a language as interactive as Applesoft Basic, yet, unlike Applesoft, you don't have to pay the price in slow execution speed. Programs written totally in FORTH are usually faster than programs written in C or Pascal and a heck of a lot smaller. Best of all, FORTH has a large library of public domain programs.

**Go FORTH** is the new FORTH language implementation for the Apple® //e, //c, //gs (//e emulation mode, full //gs version late Fall) and the Apple® ///. It is 100% ProDOS® and SOS™ supported. Go FORTH code is intercompatible with all Go FORTH supported machines machines. Go FORTH is for the hobbyist, the system developer, the applications writer, anyone who wants to learn and use the powerful FORTH language.

Go FORTH comes with its manual and an assortment of utilities in its SCREEN file. Many other utilities and support systems will be available soon. For beginners, we highly recommend the Starting Forth manual, and we recommend the Go FORTH Toolkit series for everyone!

### ONLY $69.95 Complete, order #5807

**Go FORTH** Toolkit #1 (Applesoft-like commands/utilities): $49.95, order #5809

Starting Forth by Leo Brodie (The training manual for Forth): $21.95, order #5706

Add $1.00 Shipping and handling per item.

### 24 HOUR VISA / MASTERCARD ORDER LINES

California Only: (800) 541-0900. Outside California: (800) 334-3030. Outside U.S.A.: (619) 941-5441

**PAIR SOFTWARE** (916) 485-6525

3201 Murchison Way, Carmichael, California 95608

Apple //e, //c, //gs and ///, ProDOS and SOS are registered trademarks of Apple Computer, Inc. No affiliation with Pair Software.
The anonymous variables of Leonard Morgenstem (FD VI/1) are not the only way to ease the problems of juggling the stack and writing readable code by implementing local variables. An alternative is to copy or move items from the stack to storage allocated in the word that uses them. Suppose we have a double and a single number on the stack, and that we would like them copied to local variables DBL and SNGL. We can achieve this, together with a convenient and readable syntax, by defining compiler words {, ), DBL, and SNGL so that we can write, for example,

```
: TEST ( d n -- )
{ DBL SNGL }
SNGL @ DBL @
D . . D . ;
```

where the curly brace construction causes the enclosed items to be copied to storage but not removed from the stack, and the later references SNGL and DBL push the appropriate storage addresses to the stack so that values may be fetched or stored. Since SNGL and DBL are compiler words, they can be used in the same way in other definitions without interference. How can we define the required words?

Definitions

Definitions are given in the accompanying screen listings. The word { is very simple. It is an immediate word that initializes a counter and switches from compilation state to interpret state. Local variable reference words that follow must leave their parameter field addresses on the stack and increment the counter. The word ) then has enough information to allocate the required amount of storage in the dictionary and to store the appropriate dictionary addresses in the local variable reference words, making them temporary pointers to the local storage. It also compiles the address of a primitive word ( ) that copies the stack items at run time, together with the number of items. Finally, it resets the compile state.

All that remains is to ensure that the local variable reference words DBL and SNGL, etc., push the appropriate storage addresses (now stored in their parameter fields) to the stack at run time. This is easily achieved by compiling the addresses as literals. It is most convenient to use a special defining word LOCAL to define these words. LOCAL takes from the stack a value specifying the number of cells the variable occupies, and Compiles it into the parameter field after the space allocated for the storage address. LOCAL defines the word to be IMMEDIATE and specifies its behavior during interpreting, when it must push its parameter field address to the stack and increment the word pointer; and its behavior during compiling, when it must compile the address in its parameter field as a literal. Our words above can then be defined very easily as:

```
2 LOCAL DBL
1 LOCAL SNGL
```

where there is no restriction other than stack size on the number of cells a local variable can use.

The primitive ( ) to copy stack items is given in Forth but, for speed comparable with other stack operations, it should be written as a code word in assembly language. The Forth definition assumes that the top item of the return stack points to the next item in the parameter field of the word using ( ), i.e., to the number of stack items to be copied. Not all Forth systems satisfy this assumption, and appropriate adjustments may be needed.

Forth's power is shown by the compactness of the words defined so far. However, this is an invitation to add more. One useful extension is to define a word with the accompanying primitive ( ) to move items from the stack to storage, instead of copying them. Thus, we can write

```
{ DBL SNGL -- }
```

where the syntax shows that items are removed from the stack. The definition of ( ) is identical to that of }, except that the primitive ( ) is compiled in place of ( ).

Another useful extension is to push values to the stack, like CONSTANT, rather than addresses, like VARIABLE. I have used a defining word LOCALVALUE which acts like LOCAL except that, during compilation, one of two special primitives ( ) and ( ) is compiled. ( ) is the more general, pushing a value of any cell length to the stack at run time; but it requires the length to be stored in the following byte, followed by the storage address. ( ) assumes a cell length of one, and requires only the address. Special primitives could also be defined for double and triple numbers. Again, the primitives should be code words so that references to local values will take no more time than

(Continued on page 13.)
SCR # 2
0 ( Local variables )
1 2 : ⟨⟨⟩⟩ ( ... n0 -- ... n0)
3 R@ 1+ R> C@ ( storage addr, # of 16-bit items k)
4 1+ 1 DO I PICK OVER ! 2+ LOOP ( copy items)
5 >R ( update instruction pointer );
6 7 : ⟨ ( -- )
8 0 [COMPILE] [ ; ; IMMEDIATE
9 10 : ⟨ ( pfan ... pfa1 n -- ) ( ... n0 -- ... n0)
11 COMPILE ⟨⟨⟩⟩ HERE >R 1 ALLOT ( for item count k)
12 0 ( item count ) SWAP 0
13 DO SWAP HERE OVER ! ( storage addr -> reference word)
14 2+ C@ ( # of items in variable) DUP 2 * ALLOT + LOOP
15 R> C! ( total # of items k ) ] ;

SCR # 3
0 ( Local variables )
1 2 : LOCAL ( c -- ) ( -- ) ( n -- pfa n+1)
3 CREATE 0 , C, IMMEDIATE
4 DOES> STATE @
5 IF @ [COMPILE] LITERAL ELSE SWAP 1+ THEN ;
6 7 : ⟨(--⟩) ( nk ... n0 -- nk)
8 R@ 1+ R> C@ 0 DO SWAP OVER ! 2+ LOOP >R ;
9 10 : ⟨(--) ( pfan ... pfa1 n -- ) ( nk ... n0 -- nk)
11 COMPILE ⟨⟨⟩⟩ HERE >R 1 ALLOT 0 SWAP 0
12 DO SWAP HERE OVER ! 2+ C@ DUP 2 * ALLOT + LOOP
13 R> C! ] ;
14

SCR # 4
0 ( Local variables )
1 2 : ⟨1LV@⟩ ( -- value)
3 R> DUP 2+ >R @ @ ;
4 5 : ⟨LV@⟩ ( -- value)
6 R@ C@ DUP 2 * R> 1+ DUP 2+ >R @ + SWAP 0
7 DO 2- DUP @ SWAP LOOP DROP ;
8 9 : LOCALVALUE ( c -- ) ( -- ) ( n -- pfa n+1)
10 CREATE 0 , C, IMMEDIATE
11 DOES> STATE @
12 IF DUP 2+ C@ DUP 1 =
13 IF COMPILE ⟨1LV@⟩ DROP
14 ELSE COMPILE ⟨LV@⟩ C, THEN @ ,
15 ELSE SWAP 1+ THEN ;
SCR # 5
0 ( Local variables - example )
1
2 ( Assumes floating point extensions that use the Forth stack )
3 ( Local variables are then highly desirable )
4
5 1 global v1
6 1 local v2
7 1 local VALUE n
8 decimal
9
10 : IP ( addr1 addr2 n -- r )
11 ( set inner product r of fp vectors at addr1 and addr2 )
12 ( v1 v2 n --> )
13 0EO n 0
14 do v1 @ F@ v2 @ F@ F* F+ F$:BYTES DUP v1 +! v2 +! loop ;
15
SCR # 6
0 ( Local variables - primitives )
1 ( for MM MasterForth 6502 Assembler )
2 3 code ( { } ) ( ... n0 -- ... n0 )
4 ( copy top k items from stack to local storage )
5 xsave stx ip ylabel 'A ASL N STA N 1+ STA INY
6 1 l: bot lda ip ylabel 'Y STA INX INY N 1+ DEC 1 L# BNE
7 n inc clc n lda ip adc ip sta 2 L# BCC IP 1+ INC
8 2 L: xsave ldx next jmp c;
9 10 code ( { -- } ) ( nk ... n0 -- nk )
11 ( move top k items from stack to local storage )
12 IP ylabel 'Y LDA 'A ASL N STA N 1+ STA INY
13 1 L: BOT LDA IP ylabel 'Y STA INX INY N 1+ DEC 1 L# BNE
14 N INC CLC N LDA IP ADC IP STA 2 L# BCC IP 1+ INC
15 2 L: NEXT JMP C;

SCR # 7
0 ( Local values - primitives )
1 ( for MM MasterForth 6502 Assembler )
2 3 code ( 1LV@ ) ( -- value )
4 ( push a single value to the stack )
5 IP ylabel 'Y LDA N STA INY IP ylabel 'Y LDA N 1+ STA DEY
6 CLC 2 L# LDA IP ADC IP STA 1 L# BCC IP 1+ INC
7 1 L: N ylabel 'Y LDA PHA INY N ylabel 'Y LDA PUSH JMP C;
8 9 code ( LV@ ) ( -- value )
10 ( push multi-celled value to the stack )
11 2 L# LBY IP ylabel 'Y LDA N 1+ STA DEY IP ylabel 'Y LDA N STA
12 DEY IP ylabel 'Y LDA 'A ASL TAY
13 1 L: DEX DEY N ylabel 'Y LDA BOT STA 0 L# CPY 1 L# BNE
14 CLC 3 L# LDA IP ADC IP STA 2 L# BCC IP 1+ INC
15 2 L: NEXT JMP C;
Considerations:

Variables for PROM-based Programs

Richard A. Altimus - Highland Heights, Ohio

Forth is a dictionary-oriented language. Definitions typically deal with memory locations within the dictionary boundaries. The standard treatment of variables in Forth locates variables in the dictionary, right alongside definitions of executable words and constants. Problems arise, however, when a particular application is targeted for a PROM-based system. Although this presents no problems with a majority of definitions, variables must be handled separately, or they will quickly become constants. Usually, a target system which will run from PROM will have a separate area of RAM set aside for the purpose of storing variables. The task is to evolve a system of vectoring variable operations into this RAM area.

Constraints on Method Design

Several constraints must be observed. The method must monitor RAM address allocation in order to produce a vector address into RAM which is unique to one variable, and to ensure that these addresses are within legal RAM boundaries. The method must compile pertinent parameters into the dictionary, so that they are retained in the PROM-based system. The method must produce definitions which yield, on execution, an address into RAM which is consistent with existing Forth utilities while, at the same time, approaching the variable-handling capabilities of high-level languages. This method will be relatively simple to implement while, more importantly, being consistent and flexible in use.

Specifying the Structure of the Method

Under the new method, two pointers are needed. The first pointer contains the address of the next free byte of RAM, which will be used for defining the next variable. A utility must perform the ‘allot’ function on the RAM area. The second pointer contains the address of the last usable byte of RAM, which can be used to detect boundary violations. Preferably, these two pointers are located in RAM, so the user can alter the pointer values and deal with multiple RAM areas. These pointers can be defined in two ways: as a constant (that constant being the address containing the pointer) or as a colon definition (the name of the word is the pointer name, and the definition consists of placing the address containing the pointer on top of the stack). These pointers must be initialized, by the user, to handle the RAM area in the system being used.

By storing these pointers as constants in the dictionary (prior to burning PROMs) and defining a word to restore these constants on power-up, a user can take advantage of the unused RAM area for interactive variable definition in a PROM-based system. This procedure guarantees that variables defined interactively will not interfere with previously defined variables in the PROM-based dictionary.

The dictionary entry for a RAM-based variable should consist of a standard Forth header (NFA, LFA, CFA, and PFA). Compiled into the PFA, consecutively, should be the following:

- base address vector
- number of dimensions
- individual dimensions limits
- number of bytes in each entry

These values will be retained in the PROM-based dictionary. Upon execution, the PFA is left on the stack. The subscript-handling words will construct the absolute address from the information supplied in the array reference. The opening bracket will create...
an intermediate stack that contains pertinent information for the next subscript-handling word. The subscript separator will resolve the previously given dimension and add this count to the offset, leaving the intermediate stack for the next subscript-handling word. The closing bracket resolves the supplied dimension, resolves any undeclared dimensions to zero, and produces the absolute address.

Each subscript entered in a variable reference is verified against the limit for that dimension, to prevent boundary violations. If a boundary violation is detected, an appropriate message is generated and the maximum value for the dimensions is substituted. This yields a usable address, although it is not the requested location. This also prevents altering data outside the area being accessed.

Summary

The need exists for a method of handling high-level variables which can be used in PROM-based systems. By observing the above-mentioned constraints, a method can be derived which is easy to use, yet flexible enough to meet future needs.

Richard A. Altimus is a test engineer in the Programmable Controller Systems division of Allen-Bradley.

(Continued from page 9.)

other stack operations.

Note that local variable and value references remain valid until they appear in another curly brace construction. Note also that any number of curly brace constructions can appear in the same definition, although more storage will be allocated each time. Except for simple definitions, the storage penalty for using local variables will be partly offset by a saving in stack manipulations, and the code should run faster. The biggest benefits are in ease of programming and in readability. Using lower case for local variable and value reference words further enhances readability.

Peter Ross is a research scientist who uses Forth professionally for instrument control and data acquisition, and is a member of the Australian Society of Soil Science.
There seems to be an ongoing argument between those who do and those who do not feel that Forth code is readable. It is probably safe to say that Forth allows programmers to write eminently intelligible or horrendous code. In this article, I will describe one way to achieve better readability.

The Method
One of the most powerful features of Forth is its extensible compiler. It allows us to create cross-assemblers and metacompliers, even other languages. We will focus here on the idea of creating a language within a language.

Let us assume a hypothetical case where we have a talented, software-department manager supervising a less-than-talented crew. (Of course, I have never encountered this sort of thing myself, but in an infinite universe, all things are possible.) Mr. Manager would dearly love to use the Laxen & Perry F83 package to develop software for all of his company’s projects. But he knows he will have a difficult time finding people who know how to use it.

What he needs to do is write a little, easy-to-learn language (one that, perhaps, looks like the Pascal taught in school), and place it on top of the Forth.

The Language
Before we get into an actual example, it might be a good idea to list some probable design specifications. The first thing we will need is a postfix-to-infix convertor. This will help make the listings more readable, not only for raw recruits, but also for other department heads.

The Example
There is an excellent article by Michael Stolowitz (FD IV/6) and another by Craig A. Lindley (FD VII/1,2) describing how algebraic (infix) expressions can be evaluated. The first three screens of this example are a variation on that theme, so there is no need to explain them here.

What I have added to it begins in screen 4. The defining word INTEGER is a cross between CONSTANT and VARIABLE. Actually, it is nothing more than a self-fetching variable. But in the new language, we will use it to declare single-length integers, so the name INTEGER is more appropriate.

The LET . . . NOW pair allows us to assign a value to a previously declared integer. After declaring X to be an integer, we could assign it an initial value by writing:

```
LET X = 0 NOW
```

There are a few syntax rules to be observed (even though none are being checked in this example). The first word after LET must be a declared integer, and the second word must be the equal sign. Between the equal sign and the word NOW, there must be an algebraic expression.

The algebraic expression may be a single-length literal (such as the zero above), or a combination of single-length literals and declared integers. So an expression to the right of the equal sign can be evaluated, and the result is assigned to the declared integer on the left. This allows us to set up a loop counter by writing:

```
LET X = X + 1 NOW
```

The DISPLAY word allows us to display the value of a declared integer at the terminal. It does essentially the same thing as . (dot) in Forth, except that instead of popping a value from the data stack, it takes it from the input stream. So we can display the value of X by writing:

```
DISPLAY X
```

The SHOW word does the same thing as DISPLAY, except that instead of displaying the value of X, it interprets it as an ASCII code and displays the corresponding character (like EMIT does in Forth). If X were set to 7, then DISPLAY X would display a 7 at the terminal, whereas SHOW X would beep it.

Control structures will have to be changed so that we can get away from the data stack. Don’t get me wrong... I love the data stack. But phrases like SWAP OVER DUP are notorious for adding confusion, so I would suggest avoiding its use. It is easy enough to do and well worth the effort.

All we have to do is define IF so that it simply begins the evaluation of an alge-
braic expression whose result will be interpreted as a Boolean value (true or false). THEN will terminate and evaluate the expression and conditionally branch, depending upon the Boolean result. ELSE is the same as it is in Forth, but ENDIF will now serve where Forth's THEN used to go.

For example, we might write:

```
IF X < 33 THEN
    DISPLAY X
ELSE
    SHOW X
ENDIF
```

Looping is again easily converted to an infix version. BEGIN simply marks the start of a loop as in Forth. FOREVER marks the end of an infinite loop like Forth's AGAIN.

The WHILE function is duplicated by the IF ... STAY pair. As long as the algebraic expression between IF and STAY is true, the looping between BEGIN and END will continue. For example, we could display the digits zero through nine by writing:

```
LET X = NOW
BEGIN
    IF X < 10 STAY
    DISPLAY X
    LET X = X + 1 NOW
END
```

The UNTIL function is similarly duplicated by the IF ... LEAVE pair. In this case, the looping will continue until the algebraic expression between IF and LEAVE is true. We could display those digits just as easily by writing:

```
LET X = 0 NOW
BEGIN
    DISPLAY X
    LET X = X + 1 NOW
    IF X = 10 LEAVE
END
```

(Continued from page 6.)

**F83 Execution Security**

Dear Marlin,

It is now time to look at execution security for F83 on the PC, especially because of the lack of a reset button. This works the same as before (FD IX/2). XSECUR is the patch. XSECURITY installs it, and UNSECURE uninstalls it.

Sincerely,

G.R. Jaffray, Jr.

3536 Angelus Ave.

Glendale, California 91208

---

```
HEX ASSEMBLER

LABEL XSECUR 0 [BX] JMP
LABEL XSECURITY AX LDS 89 C, C3, C, "0 [BX] AX MOV AX PUSH AX DEC AX DEC
BX AX CMP AX POP XSECUR1 JE
' QUIT @ # AX CMP XSECUR1 JE
' UNNST @ # AX CMP XSECUR1 JE
' RMARGIN @ # AX CMP XSECUR1 JE
' BL @ # AX CMP XSECUR1 JE
' BASE @ # AX CMP XSECUR1 JE
' KEY @ # AX CMP XSECUR1 JE
' EMIT @ # AX CMP XSECUR1 JE
AX BX MOV 0 [BX] AL MOV E9 # AL CMP XSECUR1 JE 103 #) JMP

CODE XSECURITY

>NEXT # BX MOV E9 # AL MOV
AL 0 [BX] MOV BX INC XSECUR
>NEXT 3 + # AX MOV
AX 0 [BX] MOV >NEXT # JMP C;

CODE UNSECURE >NEXT # BX MOV
AD # [BX] MOV BX INC
BB [BX] MOV BX INC
DD # [BX] MOV >NEXT # JMP C;

DECIMAL FORTH
```

---

**HEX ASSEMBLER**

```
HEX ASSEMBLER

LABEL XSECUR 0 [BX] JMP
LABEL XSECURITY AX LDS 89 C, C3, C, "0 [BX] AX MOV AX PUSH AX DEC AX DEC
BX AX CMP AX POP XSECUR1 JE
' QUIT @ # AX CMP XSECUR1 JE
' UNNST @ # AX CMP XSECUR1 JE
' RMARGIN @ # AX CMP XSECUR1 JE
' BL @ # AX CMP XSECUR1 JE
' BASE @ # AX CMP XSECUR1 JE
' KEY @ # AX CMP XSECUR1 JE
' EMIT @ # AX CMP XSECUR1 JE
AX BX MOV 0 [BX] AL MOV E9 # AL CMP XSECUR1 JE 103 #) JMP

CODE XSECURITY

>NEXT # BX MOV E9 # AL MOV
AL 0 [BX] MOV BX INC XSECUR
>NEXT 3 + # AX MOV
AX 0 [BX] MOV >NEXT # JMP C;

CODE UNSECURE >NEXT # BX MOV
AD # [BX] MOV BX INC
BB [BX] MOV BX INC
DD # [BX] MOV >NEXT # JMP C;

DECIMAL FORTH
```
Forth Dimensions 16

Wenrich screens:

```
0 \ FTICEAPL - OPSTK @TOS PUSH_OP POP_OP 17feb98cw \ FTICEAPL - INTEGER LET NOW 17feb98cw
1
2 vocabulary FTICEAPL
3 create OPSTK 44 allot \ operand stack
4 ; @TOS (S -- adr) \ fetch top of operand stack
5 7 OPSTK dup @ + ;
6 9 : PUSH_OP (S cfa prec -- ) \ cfa & prec to operand stack
7 4 OPSTK +* @TOS ! ;
8 12 : POP_OP (S -- ) \ drop prec & interpret cfa
9 13 @TOS 2e ~4 OPSTK *! drop
10 state @ if , else execute then ;
11
2
3 \ FTICEAPL - PREC INFIX */ + - ( ) = NOT AND OR 16feb98cw \ FTICEAPL - ?COMP DISPLAY SHOW 17feb98cw
1
2 : PREC (S -- prec @TOS @ ); \ fetch precedence from TOS
3 4 : INFIX + create swap , , immediate \ create an operator
4 does> 2e begin dup PREC > not while
5 \r \r POP_OP / / 17feb98cw \r \r repeat PUSH_OP ;
6
7 8 7 INFIX * * 7 INFIX / / \ set PREC and cfa
8 9 6 INFIX + + 6 INFIX --
9 10 5 INFIX << 5 INFIX >>
10 11 5 INFIX = =
11 12 4 INFIX NOT NOT
12 13 3 INFIX AND AND
13 14 2 INFIX OR OR
14
3
4 \ FTICEAPL - }MISSING ( ) EVAL EVAL 17feb98cw \ FTICEAPL - IF THEN ELSE ENDF BEGIN STAY LEAVE END 18feb98cw
1
2 }MISSING 1 abort* Missing )\* ; \ error cfa to patch
3 4 ( [\ ] )MISSING 1 PUSH_OP ; \ algebraic left parenthesis
4 5 \ algebraic right parenthesis
5 6 : begin 1 PREC < while
6 7 POP_OP repeat 1 PREC =
7 8 if -4 OPSTK *! else 1 abort* \begin{verbatim}
9 Missing (\* then ; immediate
10
11 \ EVAL 0 OPSTK ? ; \ begin expression evaluation
12 \ end expression evaluation
13 \ EVAL begin PREC while
14 \ begin expression evaluation
15 \ POP_OP repeat ;
```

Forth Dimensions

Volume IX, Number 4
Analysts like to proclaim the end of the low-budget, high-tech startup, but more than a few challengers disprove the rule. FD interviewer Michael Ham caught up with Lori Chavez and Derrick Miley a year after their company first released its Forth system for the Macintosh.

MH: How did you get involved in Forth?

DM: Stanford has a “smart products” course in which mechanical engineers for the Master’s year learn how to integrate mechanical systems and control them with microcomputers. That’s where, basically, our entire company came from: the Smart Products Design Lab at Stanford.

MH: Were you all in it the same year?

DM: No. Aleksey Novicov taught us, and then I taught Lori.

MH: Three generations of a Master’s Degree program.

LC: Actually four generations, because the next year they used the core of our Forth.

MH: How big is Palo Alto Shipping Company?

DM: Oh, we have about 1400 square feet! [laughs] There are three of us now. Aleksey decided that Europe was very interesting, so that’s where he is.

MH: And Terry Noyes is on sabbatical now, right?

DM: Terry just got done jamming on the Atari, and now he’s doing some consulting independently. Tim Lee is going to help us with Atari maintenance and things. Also, to get the Amiga done, we are going to work Tim into the schedule.

MH: I wanted to ask about the high quality of packaging for Palo Alto Shipping Company’s products. Who is the packaging genius in your company?

LC: What do you mean by packaging? The components of the product?

“I was ecstatic when we hit two sales.”

MH: I mean that what you sell looks like a product. It has a box, a binder, documentation — it looks complete, it looks like a real product.

LC: Our first product, for those who got it, wasn’t quite as glossy. It’s been a learn-as-you-go process. We didn’t know how to make a manual, how to put a package together, and what you saw at this show was a year’s worth of knowledge. It came together nicely. It took us a while to realize that the binder format was the way to go.

MH: Why is that the way to go?

LC: Well, it’s the way to go if you can afford it. At $49.95 we couldn’t — we had to do the manual’s binding like a paperback book. A binder is the way to go, because you can keep the customers updated on documentation. When you print the manual as a paperback, you pretty much have to buy a new version whenever there’s a change.

DM: And Apple changes. The Mac is our big product and it changes almost monthly. They’ll bring out a new manager and surprise everyone. Or a new machine.

MH: How has Apple been about keeping you vendors informed?

DM: We have a line to Apple, we get all their technical docs, and we’re always there. They have a mechanism that allows us to get the documentation we need.

MH: How did you decide to target the Mac?

DM: Availability. And we didn’t want to compete with the company targeting the PC.

MH: And it’s getting hot again.

LC: Well, it was a hot, new machine then. And, being at school, we found a lot of Macs. Apple really pushes it at the university.

MH: The Mac seems to be on the move again, and they’re not going to let it die.

DM: I also had PDP-11 background; that was my first assembly language. And when...
you see PDP-11 and then you see 68000, you just know you don’t want to go with Intel.

MH: Three graduate students fresh out of school starting a company — interesting. Did any of you do other work outside the university before starting this company?

DM: We met at a start-up in Fremont funded by Dysan. They were developing a new Forth, and that was where we learned Forth and how to write it. And we saw limitations in the product we were using. We saw there was a way to do an even better implementation; that’s where we learned how to do it and how to do it better.

MH: That was before the Stanford course, or after?

DM: After Stanford.

LC: A year after graduating.

MH: How was the company born? One night you were sitting around saying, “Hey, I’ve got a garage, and you’ve got a computer; we could get together and put on a real show!”

DM: Lori and I were graduating, Terry came back from a winter in Germany, and we just came together and said, “Let’s do it.”

MH: Was there a historic placemat, as with Compaq?

DM: We were all sitting around a queen-sized bed—

LC: —crammed in a dorm room—

MH: How did the jobs parcel out among the four of you in the start-up year?

DM: Aleksey was the vision. He and I would bat around what it was going to be. He did the research and looked at everything that we would do. And Lori did the documentation; the whole manual was written by Lori. Terry and I did the grunt work — we did the coding! I did more Forth, he did the assembler and the debugger.

MH: Interesting that there was no marketing person, per se.

LC: And our marketing presence, I think, shows that.

DM: In terms of where we are going, that’s the final step.

MH: If you had it to do over again at the company, knowing what you now know, what would you do differently?

LC: I think we would not have initially come out at the low price we did. It was very altruistic, and coming out of school, our thought of marketing was, “low price, high sales.” But you learn that there are many more factors than price involved in whether a person buys your product. In fact, price has very little to do with it. That’s the major thing.

MH: Rick, what would you do differently?

DM: The pricing; Lori answered the big one. But also running ourselves too thin — don’t spread yourself too thin.

MH: In the number of processors?

DM: No, we never really switched processors — we stayed with the Motorola. It was the number of computers we tried. When you only have two or three people, you can’t afford to lose three months on any project. You have to ship. When the Atari and the Amiga came out, we tried to get them all. We came up short on the Amiga.

MH: Probably the right one to come up short on.

DM: Yes, it would have been better just to have all of that work back.

MH: To spend on another machine.

DM: Or anything else.

MH: Had you done any programming before the Smart Products Design Lab, Lori?

LC: Just college courses: 68000 programming, Pascal. No real programming, never
on a microcomputer.

MH: And it sounds like only in a class environment, with assigned problems and such.

LC: You find out that real programming is very different.

MH: You are all mechanical engineers?

DM: Yes. Lori doesn’t have a Masters, she’s the baby.

MH: What is your future direction? Let’s start with where you are now. You have a product for the Mac, and it’s now version 2.0, which is Mach 2.

DM: Another thing we’d do differently is not pick names that people sue us for. We got into litigation over the name Mach 1, so we had to switch to Mach 2. That has caused confusion in the marketplace.

MH: Not too bad. Perhaps you should let Mach 2 mean version 2.0, and when you come out with version three, call it Mach 3. I thought it was a clever ploy, that every new version was going that much faster.

DM: Except that we were advertising. We had labels of Mach 1 and product of Mach 2, and ads...

MH: Is your Atari version released now?

DM: It’s out.

MH: And your Amiga product?

DM: That’s not shipping yet.

LC: I think the biggest thing that’s happened this year is that people know the name of Palo Alto Shipping.

MH: How did you get the name?

DM: Detroit Diesel was taken.

MH: Right. Now, how did you get the name?

DM: We were sitting around on the bed and we wanted a generic name. We were all just gabbing away.

MH: Seemed like a good idea at the time?

DM: Yes, and it is a good name. People think we’re much larger than we are.

LC: Everyone has to ask why we have it, so for that reason it sticks in their mind. It’s so — odd.

MH: I certainly agree that “Palo Alto Shipping Co.” is established now. How do you make yourself known?

DM: MacWorld.

LC: For shows and conventions, we try to stay in the area.

DM: We’ve gone to Chicago for conventions, and Silicon Valley conventions are big. A lot of Mac conventions are out here.

MH: Is the 68000 your future? Any 68000 machine that comes out, you’ll at least look at?

DM: Right. We’re on the OS-9, we’re on the CD-I stuff that we’re hoping is a market, and the EPROM version.

MH: You have a target compiler that people can order and get now?

DM: Yes.

MH: What is your product line?

DM: Macintosh, number one, biggest; Atari, smaller; and OS-9; and EPROM, the target compiler.

MH: What is the OS-9?

DM: It’s a multi-tasking operating system from a company in Iowa. It’s the operating system that was chosen by Sony and Philips for CD-I (Compact Disk—Interactive). It’s like a simple Unix.

MH: You don’t see any departures from the 68000?

DM: The IBM market is attractive, but it’s
very competitive.
MH: I think the opening ante in that market is fairly substantial now. When did Mach first hit the market?
MH: A Christmas present for people.
DM: We barely had it done; I don’t think we’d delivered it.
MH: What other goals do you have?
DM: We want to make Forth more mainstream.
MH: In what way?
DM: We’re advocates of having text files, infix assemblers, and standard debuggers and linkers. Our goal is to have Forth be an interactive environment. That’s its strength. And then on top of that, use all the other strengths of all the other languages. I want to be a Forth programmer who can converse with C programmers and not be ostracized or put off.
LC: It seems as if the emphasis today is going away from what language you use — C, Pascal, or Forth — and toward how good your environment is. Other languages are very powerful and give access to things you didn’t have before, but the big thing now is your environment: how nice it is, how well the tools are integrated, how fast it gets work done, how much it helps you. Forth doesn’t usually have that much of an environment around it. So we’re trying to add that environment, which other languages seem to have as a standard.
MH: In almost every company, particularly among start-ups, there is a particular black moment, a moment when you say, “This is not going to work.” Did you have a black moment?
DM: I’ve been with start-ups since I was seventeen. I’ve had a sound and light show, I’ve installed stage lighting, all contract work. I’ve had a flight-case company, I have been manufacturing as a single person for eight years. This is nothing. Black moments are—
MH: —a dime a dozen.
DM: You see, for me, I’ve worked many years and then sold one. So I was ecstatic when we hit two sales.
MH: I see; the secret is to have a low threshold of satisfaction.
DM: Well, I’d been in business, as a young person with the drive to be in business. This is phenomenal success for me, even if from an engineering or accounting perspective it is mediocre or survival. To me, survival is success. If I can sell in singles, I’m happy.
MH: What’s a flight case?
DM: A heavy-duty metal transport case. I was into sound and light shows, and—
MH: —you needed something to carry them around in.
DM: Right, it was a natural evolution. I was getting reamed at the stores, so I started making them.
MH: So how did you get this entrepreneurial bent?
DM: I’ve got brain damage.
MH: [laugh] No, no, come on now. That’s evasive.
DM: It’s inside of me. It’s not something—
MH: Not from a parental example, or a competitive brother or a teacher who said, “Young Rick, I want you to...” Just something on your own?
DM: Yes, it’s what I have; it’s the ability to never give up. I could say, as long as it pays the rent, I’m in the game.
MH: It seems that one needs two things. One is persistence. That’s the one you just described. The other is initiative, taking the step that wasn’t required. Stepping up above where you are.
(Continued on page 37.)
Ne of the things no developer of a complete floating-point math package can escape is the horror of designing transcendental functions. While writing my book MVP-FORTH Integer and Floating Point Math, I found myself faced with the dilemma of how to implement quick, accurate, and relatively compact math functions (optimizing for any two of these qualities is easy — all three at once is no fun at all).

My first step was a simple engineer's reflex: I picked up my CRC math handbook. This handbook has a complete set of formulas for implementing almost every function that exists. However, there was a slight problem. Watching a Taylor-Maclaurin expansion for arc-tangent converge is like watching grass grow. I decided there had to be a better way.

After surviving a numerical analysis course and many hours of research in the dusty corners of a university library, I emerged with a set of equations that I believe are relatively efficient, accurate, and compact. I have not presented derivations for these formulas, as each equation takes at least an hour and many sheets of paper to derive. All the equations given here are designed to be used with 31-bit mantissa intermediate calculations, and produce an accurate (seven to eight decimal digit), 24-bit mantissa result with the minimum number of required terms. Readers who desire different precisions (such as 16-bit scaled integer), or who want to understand the theory behind these equations, are invited to wade through a book like Numerical Analysis, and books on computer arithmetic, such as Computer Approximations, Mathematical Functions and Their Approximations, and Mathematical Methods for Digital Computers.

**SIN(X)**

Sine is the basic trigonometric function. The equation given is a Chebyshev polynomial, and is valid on the range \(-\pi/4\) to \(\pi/4\) radians.

\[
\sin(X) = 0.9999999995 + 0.1666666663 \cdot X^2 + 0.008333328785 \cdot X^3 - 0.0001983920268 \cdot X^4 + 0.000002717349463 \cdot X^5
\]

**COS(X)**

The cosine equation given is a Chebyshev polynomial, and is valid on the range \(-\pi/4\) to \(\pi/4\) radians.

\[
\cos(X) = 1.0 - 0.4999999943 \cdot X^2 + 0.0416666167 \cdot X^4 - 0.001388661862 \cdot X^6 + 0.0000243798031 \cdot X^8
\]

**ATAN(X)**

Arc tangent is the primary inverse trig function, and may be heavily used by graphics applications. Series approximations for this function can be painfully slow when the range is near \(\pm 1\) (and the resultant angle is near 45°). The equation given is a Chebyshev polynomial, is valid on the range -1 to 1, and gives a result in radians.

\[
\tan^{-1}(X) = 0.9999999842 + 0.333306679 + 0.1999248354 - 0.1420257041 + 0.106367506 \cdot X^5 - 0.0749544546 + 0.0425876076 \cdot X^9 - 0.0160050306 + 0.0028340643 \cdot X^{13}
\]

**LOG2(X)**

Log base 2 is a crucial function for calculating any logarithm, and for output formatting. The equation given is valid on the range 0.5 to 1.0, and was derived by taking the Taylor-Maclaurin expansion for \(\ln(x)\) and dividing it by \(\ln(2)\). The choice of base 2 as the primitive logarithm reduces the number of terms in the expansion by limiting the range, and provides for efficient range reduction in a binary-exponent, floating-point package.

\[
\log_2(X) = 2.885390082 + 0.9617966939 + 0.577080164 + 0.4121985831 + 0.3205988980 + 0.2623081893 + 0.2219530832 + 0.1923593388 + 0.1697288283
\]

\[2^X\]

A primitive exponential function is required to complement the logarithm function, and is useful for output format-
ting. The equation given is a Chebyshev polynomial, and is valid on the range 0.0 to 1.0.

\[ 2^x = \\
 1.0 \\
+0.69314718 \quad x^2 \\
+0.24022636 \quad x^3 \\
+0.055505294 \quad x^4 \\
+0.009613538 \quad x^5 \\
+0.0013429811 \quad x^6 \\
+0.00014299401 \quad x^7 \\
+0.000021651724 \quad x^8 \]

Range Reduction

By now, you have noticed that the equations given have restricted ranges for the input variable X. This limits the number of terms in the equation, while keeping accuracy high. Simple trigonometric and algebraic identities can be used to force all inputs into the required ranges. Some of the more useful identities are given below (all arguments are in radians):

- \( \sin(x) = \sin((x+2n) + 2\pi) \)
- \( -\sin(x) = \sin(-x) \)
- \( \cos(x) = \cos(-x) \)
- \( \sin(x) = \cos(x-\pi/2) \)
- \( \log(x) + \log(y) + \log(2) \)

Derived Functions

The formulas given in this article may be used to derive all desired transcendental functions. The CRC handbook gives useful identities for creating any desired function. Special care must be taken when using these identities, to avoid exceeding range limitations and to avoid division by zero.

- \( \tan(x) = \sin(x) / \cos(x) \)
- \( \sec(x) = 1 / \cos(x) \)
- \( \csc(x) = 1 / \sin(x) \)
- \( \cot(x) = \cos(x) / \sin(x) \)
- \( \arcsin(x) = \arctan(x / \sqrt{1-x^2}) \)
- \( \arccos(x) = \pi/2 - \arcsin(x) \)
- \( \arccot(x) = \pi/2 - \arctan(x) \)
- \( \arccsc(x) = \arctan(\sqrt{x^2 - 1}) \)
- \( \arccsc(x) = \arctan(1 / \sqrt{x^2 - 1}) \)

Logarithms and exponentials use the formula:

\[ \log_a(x) = \log_b(x) / \log_b(a) \]

\[ \log_10(x) = \log_2(x) / \log_2(10) \]

\[ \ln(x) = \log_2(x) / \log_2(e) \]

References

   (This is a complete, machine-independent mathematics package written in MVP-FORTH. The equations presented in this paper are the same approximations used by the math package included in this book.)

   (This old standby is filled with useful mathematical formulas and tables. It should be owned by anyone interested enough in mathematics and/or computers to get this far in the article.)

   (This is a college textbook full of algorithms for numerical methods and approximations.)


   (This book contains pre-computed, 20-digit coefficient tables for just about any function you would care to approximate.)

Phil Koopman, Jr., is the vice-president and chief engineer for WISC Technologies.
Possibly, the logic most learned on university campuses is truth-functional logic. What makes this attractive to undergraduates and their teachers is exactly what makes it attractive to the programmer; namely, that it utilizes a mechanical decision procedure for testing logical validity.

This procedure is based on a matrix called a "truth table."

**Truth-Functional Logic**

Logic, in general, has only one objective. That is to determine whether arguments are, or are not, valid. We say that an argument is valid when it is impossible, short of embracing a contradiction, for its premises to be true while its conclusion is false. This criterion is often picturesquely expressed by the dictum that an argument is valid if and only if the conditional that corresponds to it is true in "all possible worlds."

When using truth-functional logic, it is possible to give this way of putting it a very concrete form. Truth-functional logic is, in a sense, the least sensitive logic, taking as its smallest particle the unopened sentence and testing compounds of sentences to see if they could possibly be false. Since the logic stops at the level of the sentence, the content of a sentence is of no interest, only its truth-value.

However, truth value in the actual world is also of no interest, for logic's only concern is whether there exists a pattern of possible truth values for a set of sentences in a logical compound, such that the compound itself is false. This leads us to the connectives used in forming compounds.

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p → q</th>
<th>(p → q) &amp; q</th>
<th>[ (p → q) &amp; p ] → q</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

**Figure One. Modus Ponens:** If ((if p then q) and p) then q

called the truth-functional connectives. These connectives, including AND, OR, and IF THEN, among others, are themselves defined in terms of patterns of truth-values. The connective OR, for example, is defined as follows:

\[
p \lor q
\]

The p's and q's stand for any propositions whatever, and the matrix is an example of a truth-table (here used not to test for validity, but to define). The four rows under p and q generate all the possible worlds for two propositions, and the column under p OR q defines the OR connective for each of the four worlds: OR compounds are true in all worlds except those in which both (or all) its disjuncts are false. Once the connectives have been defined, a compound based on those connectives can be evaluated for truth-value; if it evaluates as true in every distribution of truth values across its constituent propositions, then it is a tautology and the argument that corresponds to it is valid. The argument form called *Modus Ponens*, for example, which reads "If ((if p then q) and p) then q," is shown in Figure One to be valid.

With the exception of languages such as Prolog and Lisp, the coding of a logical, validity-testing algorithm presents intimidating obstacles. While most languages support at least a minimal set of logical operators, typically NOT, AND, OR, and XOR (a few having also IMP and EQV), they are there primarily in the interest of allowing compound tests; it is not obvious how they could be easily adapted to the evaluation of statements in the propositional calculus.

It is important to note, initially, the difference between the IF THEN supported by all languages and the IMP supported by some; the IF THEN is, in fact, what I'll call an "executive" conditional, in view of the fact that its consequent is an operation to be executed, not a proposition. IMP is the operator found in propositional logic, where it occurs as "material implication" and is commonly represented by the "horseshoe" symbol (I have used - > above).

When approaching the problem ini-
having no more than five variables. The integers will be so chosen as to create a truth table array, thereby generating all possible worlds for five elements. While the code could be made general enough to allow the program user to decide the size of the truth table, there are coding obstacles — such as the absence of a square-root function — that make such efforts not worth the prize. The best approach for the user is to code, once and for all, the largest case and use that code for smaller cases as well.

This is intuitively satisfying because the conjunctions themselves are Boolean, capable of bearing truth value. In non-symbol-processing languages, the only Booleans are those based on the arithmetic comparison operators, e.g., =, <, >, etc. Since we are taught to use the connectives with these Boolean conjunctions, and since this is intuitively well supported, we tend to forget that their use in this situation is, in fact, derivative. The logical connectives do their work natively, not at the macro but at the micro level; not at the level of Booleans, but at the level of bits. And, for example, typically (in an eight-bit environment) compares a two-byte value with another two-byte value on a bit-by-bit basis, generating a new two-byte value whose bits are 1, where those compared were both 1 and 0, where at least one of those compared was 0. As an example, let us consider the decimal numbers 480 and 3278 connected by XOR. The truth-functional definition for XOR illustrated in Figure Two.

The two, two-byte values compared are typically returns from the evaluations of Booleans, but they need not be. They could have been any integers at all. In a word, the logical connectives actually operate on integers, and they do so at the bit level.

But! If it is really integers that we logically connect, and integers can be made to have appropriate bit patterns, then using appropriate integers in propositional formulae is the same thing as running them through a truth table. The point may, perhaps, be best seen by looking at the actual integers and their binary representation in Figure Three.
Tilting one’s head to the right (looking at the binary rows as columns) reveals a standard truth table for five variables. To check whether a formula is a tautology, contradiction, or contingency, we simply enter the formula, using as propositional variables double-number variables whose values are the above numbers with 32-bit versions of the logical connectives binding them together. Forth will evaluate the formula and yield either 8589934591 (all 1's), 0 (all 0's), or a number in between. As might be expected, 8589934591 implies tautology (true in all possible worlds), 0 implies contradiction (false in all possible worlds), and between implies contingency (true in some worlds, false in others).

The Code

Screen 1 contains all the variable declarations, as well as a mixed mode multiplication operator \( \text{D} \times (\text{d n} - \text{d}) \) borrowed from Alan Winfield’s excellent book, *The Complete Forth*, and a double = operator. It should be noticed that the variables q, r, s, t, and u, which will hold the propositional variable values, are allotted four bytes each, since they have to hold double-precision numbers.

Screen 2 holds defined, 32-bit logical connectives (\( \neg, \vee, \wedge, \neg\neg \)) and five short words, q1 through u1, included only to make the entered formulae easier to read by making the @ unnecessary.

Screen 3 contains, first, TVCOLM, which generates the double integers listed earlier, and which could be expanded to yield a similar set of numbers for 64 bits. This enterprise would involve not only machinery for representing numbers of that size, but all of the supporting functions as well: not only the bigger multiplier and the bigger connectives, but a bigger \( \text{D} \), \( \text{T} \), \( \text{DUP} \), and so on; items supplied already for this code, since Forth substantially supports double numbers.

The task is not impossible, but if undertaken, perhaps it should be done with numbers much larger than 64 bits in mind, since moving to 64 would only add one more propositional variable or column. Should one choose to set the program up for 10 variables, one would need numbers 1024 bits long. The ten numbers being logically connected would jointly occupy 1280 bytes of memory, plus another 128 bytes for the evaluation column itself. While this may seem a lot, consider that doing the same with a byte array would use 10240 bytes plus another 1024 for evaluation; 11264 bytes vs. 1408. The other word, ?V, simply checks the top of the stack for the result, yielding the appropriate screen printout for the formula in question.

One sets the stage for an evaluation by running TVCOLM, which not only generates the five numbers, but stores them in the variables q through u. After that, it is only a matter of noting that we will use the words q1 through u1 instead of the variables q through u, the former automatically leaving their values on the stack, and becoming accustomed to reverse Polish, or postfix, notation. Most philosophers were taught their logic in algebraic notations, but RPN is not totally unknown, is not difficult to master, is easier for the machine to digest, and may, for all that, be aesthetically superior. At any rate, notations come in three unsurprising options: prefix, found in Polish Notation and Lisp; Infix, found in algebra and BASIC; and postfix, found in Forth and on Hewlett-Packard calculators. Using the 32-bit connectives defined on screen 1, Hypothetical Syllogism is entered as follows:

\[
q1 \text{ r1 -> r1 s1 -> } \& \text{ q1 s1 -> ->}
\]

Evaluation proceeds: q1 and r1 go on the stack, two double numbers taking up four sixteen-bit locations; they are IMPed by \( \rightarrow \), leaving one double number on the stack; r1 and s1 are pushed onto the stack and we get six sixteen-bit locations taken up until the last two are IMPed and we are back to four; the two doubles are ANDed by \&\, leaving again one double number. The IMPing is repeated for q1 and s1, leaving a total of two doubles, IMPed again for one double number remaining on the stack. If the trick works, the remainder should be 8589934591, or 32 binary 1's, indicating a tautology.

Conclusion

Once it is realized that the key to the above procedure lies in generating the integer whose binary form has the bit distribution appropriate to the left-hand column of a truth table for the given number of variables, one also realizes that a language which handles larger integers is more adapted to our purpose. It should now be apparent that the choice of Forth for coding this program was not casual; no other language could have provided the capability for creating integers of any size, the operators for manipulating them, and the debugging ease and extensibility that was necessary. Where plasticity is called for, Forth is unquestionably the language of choice; as logicians once were wont to say at the end, Q.E.D.

Jean-Pierre Schachter is the Dean of Arts and Social Science at Ontario’s Huron College.
( Scr 1 - TV alg for 5 vars using double numbers )

Variable Eu Variable El Variable Fu Variable F1
Variable q 4 allot Variable r 4 allot Variable s 4 allot
Variable t 4 allot Variable u 4 allot Variable H 4 allot
Variable N 8 allot

: D* s->d Eu ! El ! Fu ! F1 ! ( d n __ d )
   El @ Fu @ u* ( mixed-mode * operator )
   El @ Fu @ u* drop +
   Eu @ F1 @ u* drop +

: s~ 65535 xor ; ( 16 bit bit-not )

: D= rot = rot rot = * 1 = if 1 else 0 then ;

( Scr 2 - TV alg for 5 vars using double numbers )

( 32 bit logic operators )

: xv rot xor rot rot xor swap ; ( xor )
   rot xor rot rot xor swap ; ( xor )
   ~ swap s~ swap s~ ; ( bit not )
   & rot and rot rot and swap ; ( and )
   v rot or rot rot or swap ; ( or )
   -> rot s~ or swap rot s~ or ; ( imp ant cons ___ )

( SCR 3 - TV alg for 5 vars using double numbers )

: TVCOLM 65535. ( makes 5 ints, 1 for each column )
   2dup H 2! 2 4 16 256
   8 0 do N I + ! 2 +loop
   8 0 do
     H 2@ 2dup N I + @ D* xv`2dup H 2!
     2 +loop
   q 2! r 2! s 2! t 2! u 2! ;

: ?V 2dup 8589934591. D= if ( outputs decision )
   ." TAUTOLOGY - VALID" drop drop else
   0. D= if ." CONTRADICTION - INVALID" else
   ." CONTINGENT - INVALID" then then ;
Would you like to use all Forth words from the interpreter—including words like DO, IF, and BEGIN? Without having to redefine them? Then just patch QUIT and \?COMP as described below.

First of all, you have to reserve a block of memory:

\begin{verbatim}
THERE ( -- adr )
address of reserved memory
SIZE ( -- #bytes )
number of bytes reserved
\end{verbatim}

It really doesn't matter how you do this, but there are two obvious ways:

\begin{verbatim}
512 CONSTANT SIZE
\end{verbatim}

In this case, you put the field in the word list. Another way is to permanently decrease the number of screen buffers by one. In my system, that could be done like this:

\begin{verbatim}
1028 CONSTANT SIZE
FLUSH SIZE MINUS LIMITS +!
LIMIT CONSTANT THERE
\end{verbatim}

In this memory, all nested blocks of words that begin with a word containing \?COMP will be precompiled and executed when interpreted! Now you can do conditional loading (as in screen 3) and alternative loading (as in screen 4).

The principles are:

(1) every leading structure word (e.g., DO, IF, and BEGIN) starts with \?COMP, and
(2) every leading structure word pushes a value on the stack (to be checked by \?PAIRS).

The PRECOMPILE routine compiles every word from the first structure word to the last, and the last structure word pops the stack back to the initial level, thereby exiting the WHILE loop in PRECOMPILE.

When a structure block has been compiled THERE, PRECOMPILE makes it execute by pushing THERE on the return stack. PRECOMPILE, itself, runs every time a word containing (the patched) \?COMP is interpreted.

There are no special limitations to nesting different structures, since the code is

(Continued on page 36.)
scr #2

0 ( Interpretive fig-FORTH )
1
2 : PREPARE ( -- ) THERE
3 R> R> ' PRECOMPILE >R >R >R ;
4
5 : (?COMP) ( -- ) STATE @ 0=
6 IF PREPARE THEN R> DROP ;
7
8 : (QUIT) ( -- ) THERE? IF OLDP @ DP ! THEN ( new part )
9 0 BLK ! [COMPILE] I ( old QUIT )
10 BEGIN CR RP! QUERY INTERPRET
11 STATE @ IF ." ok" THEN
12 AGAIN ;
13
14 ( And now - the patch! )
15 ' (QUIT) DUP NFA FENCE ! CFA ' QUIT ! ' (?COMP) CFA ' ?COMP

scr #3

0 ( Load screen ) DECIMAL
1
2 -FIND TEST IF DROP DROP ELSE 40 LOAD THEN
3
4 50 40 DO I LOAD LOOP ;
5

scr #4

0 ( Menu screen ) DECIMAL
1
2 ." MENU* CR CR
3 ." 1 alternative A* CR
4 ." 2 alternative B* CR
5 ." 3 alternative C" 2 SPACES
6
7 BEGIN KEY CASE ASCII 1 OF 10 LOAD ENDOF
8 ASCII 2 OF 20 LOAD ENDOF
9 ASCII 3 OF 30 LOAD ENDOF
10 DUP OF 7 EMIT 0 START ! ENDOF
11 ENDCASE START @
12 UNTIL CR CR START @ EXECUTE ;
13 It is a good idea to start the execution of the loaded
14 words outside the precompiled block! Then there is no
15 importante code left THERE.
EXTENSIONS FOR F83

ANTHONY T. SCARPelli - PORTLAND, MAINE

This article explains a number of new screens I created for F83. They cover a wide range of subjects. Two are a revamp of the screen checksum calculation presented in Forth Dimensions some time ago, but here are modified to compensate for the special words of F83. Others include a special set of words to access some of the BIOS and PC DOS interrupts. There are also screens that allow you to set and print the date and time, so they can be inserted into your index and screen lists.

One of the first things we need is a set of words that can utilize the BIOS and DOS interrupts. In this way, we have at our disposal a host of routines for most any purpose. I did not want to use the few simple interrupt calls available in F83, but a set that would be more versatile.

In screen #1, we have created our first BIOS word, INTCALL. It is not your usual BIOS call word, since it looks like it only calls interrupt 0. But we don't really use it for that purpose. The 0 is actually the location where we install any interrupt number we want. So, first, we create a variable INTADR to hold the address of that location in the word. Next we create a code word INTCALL that pops four numbers off the stack and puts them into their respective registers. We save the two registers, SI and BP (by necessity), and then see the 0 INT.

This is where the interrupt number will go. It is nine bytes from the beginning of the code field address. We then return BP and SI, and push all the registers back onto the stack.

To use this routine, we merely put all the required register values on the parameter stack, as well as the interrupt number. What is left, after the word is used, are all the registers — intact, so that any one or more returned values can be used.

This, then, is our basic word to call any BIOS interrupt.

The routine in line 9 finds the address of INTCALL, adds nine to it, and saves it. Finally the word BIOSINT is created. It merely finds the interrupt number on top of the stack and places it in the byte before the INT instruction. Thus, we have made a universal call word that can handle almost any interrupt need.

To show this, screen #2, creates some general routines that all have a similar structure; they only differ in what they leave behind. The use of any of them is determined by which interrupt you are using, and which registers you need returned.

The word FUNCREQ in line 12, for instance, is used for the BIOS call 21H, that requires a function number in the AH register, and some value in the DX register. It leaves the AX register on the stack.

As some other examples, I've created some more handy words that demonstrate how these interrupt words can be easily used. Screen #3 has two very handy words that get the time and date. The word GETTIME in line 4 first calls function request 2CH via interrupt 21H. It leaves the time on the stack, which can be cleaned up for various uses. Some later words show how this can be done.

The word GETDATE in line 9, in a similar manner, leaves the date.

There are two words in the next screen that either must be in your dictionary, or they must be loaded in with their own screen. They are "MONTH and "DAY. They are found in F83's CLOCK.BLK, screen #2. The word that creates these arrays, "ARRAY, must also be available.

Screen #4 uses the GETTIME and GETDATE words to form two more usable words. The word (DATE) in line 2 first gets the date, prints the month (from the "MONTH array), and then prints out the day, a comma, and then the year. The word DATE in line 12 merely adds a space. Thus, this word can then be used when printing out a listing of screens.

The word (TIME) works in a similar manner, but first determines whether the time is after noon or not, then adjusts for 12-hour timekeeping. The hour is printed, a colon, and then the minute (after we add a zero if the number is less than ten). This is just to keep the alignment correct. Finally we determine whether we are in the a.m. or p.m. with the word in line 5, and print it. The word in line 13, TIME, is the general-use word and adds a space. This word, and the DATE word above, make documenting screen and index listings a breeze. No longer do you need to hand-print the date and time.

To show how the date and time can be easily added to an index listing, screen #5 shows my word called PINDEX (Print Index). The word on line 2, ITITL (Index title), first prints the file name, and then the date and time. The PINDEX word on line 5 includes my word PRINT, which turns on the printing command. It is defined as PRINTING ON. The word CRT is defined as PRINTING OFF. These words go back to when all I used was MMS-FORTH. It's hard to change some habits that seem so logical. The word FF is a form feed, and is
defined as \texttt{12 EMIT 12 EMIT}.

These words are all right as long as you have already set the date and time from DOS. But if you haven't, you need some words to set the time and date from Forth. That is done with the two words in screen \#6.

The word \texttt{INPUT?} in this screen is also from F83's \texttt{CLOCK.BLK}, screen \#4, and is used to get the numbers we need for the set words. If you have this word already in your dictionary, you won't need to load it; otherwise, place it at the beginning of this screen.

The word \texttt{SET-DATE} is a continuous loop that won't exit until you input the correct form for the date. That is, as an example, you can't enter 13 for a month or 32 for a day. In line 4, after the \texttt{BEGIN}, we set up some of the parameters for the interrupt. We next get the year, month, and day, and adjust them for interrupt 33D, function 43D. This interrupt leaves a zero for a valid date, and an FFH if the date is invalid. We then will get an error message if we don't put in the right data.

The word in line 10, \texttt{GET-TIME}, works in a like manner; however, it waits for you to press a key to set the time. This allows for a more accurate setting of the time, which can be down to the second.

There is one other time word I find useful. That word allows us to time intervals. To do that to a greater degree than the \texttt{TIME} word, we use all the stack values that are left by the \texttt{GETTIME} word. This allows accuracies to 1/100 second. The word is in line 2 of screen \#7. Of course you have to allow for various timing inconsistencies in any use of this word, but it will allow for the timing of the execution of loops, words, etc.

The next word I had to develop, on a lower level, was for stopping the system when it was executing various words. That is, I needed a true Control-Break routine that would cause a jump to the warm-start word. The screen \#8 shows how this was done.

F83's warm start can be executed by either typing the word \texttt{WARM}, or it can be jumped to via a vector that is located at offset 0103H. The jump has to be done with machine code, so in line 10 we create an interrupt routine called \texttt{INTRTN}. We use the \texttt{LABEL} word to create it, so the address of the routine is left on the stack when we use it.

![Figure One. Checksum flow chart.](image-url)
Screen # 1  crc ver = 17675

0 \ BIOS INTERRUPTS -- intcall
1
2 HEX
3 VARIABLE INTADR
4 CODE INTCALL DX POP CX POP BX POP AX POP
5 SI PUSH BP PUSH 0 INT
6 BP POP SI POP
7 AX PUSH BX PUSH CX PUSH DX PUSH
8 NEXT END-CODE
9 ' INTCALL 9 + INTADR !
10
11 : BIOSINT ( ax bx cx dx int# -- ax bx cx dx )
12   INTADR @ C! INTCALL ;
13
14
15 DECIMAL  -->

Screen # 2  crc ver = 23233

0 \ BIOS INTERRUPTS -- int0,1,2,3, funcreq
1
2 HEX
3 : INTO ( ax bx cx dx int# -- )
4   BIOSINT 2DROP 2DROP ;
5 : INT1 ( ax bx cx dx int# -- ax )
6   BIOSINT 2DROP DROP ;
7 : INT2 ( ax bx cx dx int# -- cx dx )
8   BIOSINT ROT DROP ROT DROP ;
9 : INT3 ( ax bx cx dx int# -- ax bx )
10  BIOSINT 2DROP ;
11
12 : FUNCREQ ( func# dx -- ax : int21 )
13   SWAP 100 * SWAP 0 0 ROT 21 INT1 ;
14
15 DECIMAL  -->

Screen # 3  crc ver = 47261

0 \ DOS FUNCTIONS -- gettime & getdate
1
2 HEX
3
4 : GETTIME ( -- 1/100sec seconds minutes hours )
5   2C 100 * 0 0 0 0 21 INT2
6   DUP 00FF AND SWAP 100 /
7   ROT DUP 00FF AND SWAP 100 / ;
8
9 : GETDATE ( -- year day month )
10   2A 100 * 0 0 0 0 21 INT2
11   DUP 00FF AND SWAP 100 / ;
12
13 DECIMAL
14
15 -->

Volume IX, Number 4  31
Forth Dimensions
Screen # 4  crc ver = 40071

0 \ date & time 06JAN86ats
1 2 : (DATE) ( -- date ) GETDATE
3 1- "MONTH TYPE SPACE (U.) TYPE ", " (U.) TYPE ;
4 5 : ?AM/PM ( -- ) 11 >
6 IF ." pm" ELSE ." am" THEN ;
7 8 : (TIME) ( -- time ) GETTIME
9 DUP DUP 12 > IF 12 - THEN (U.) TYPE ":" SWAP
10 DUP 10 < IF ." 0" THEN (U.) TYPE ?AM/PM DROP DROP ;
11 12 : DATE ( -- mon day, year ) (DATE) SPACE ;
13 : TIME ( -- hour:min am/pm ) (TIME) SPACE ;
14 15 -->

Screen # 5  crc ver = 43911

0 \ EXTENSION WORDS -- pindex 09FEB86ats
1 2 : ITITL CR ." INDEX FOR: " FILE?
3 20 SPACES DATE SPACE TIME ;
4 5 : PINDEX ( from to -- )
6 PRINT ITITL CR INDEX FF CRT ;
7 8 9 10 11 12 13 14 15 -->

Screen # 6  crc ver = 25080

0 \ DATE AND TIME -- set-date, set-time 09FEB86ats
1 2 : SET-DATE ( -- ) BEGIN 43 256 * 0
3 CR ." Year? " INPUT? DROP
4 CR ." Month? " INPUT? DROP 256 *
5 CR ." Day? " INPUT? DROP OR 33 INTl 255 AND
6 IF CR ." Invalid date!" 0 ELSE -1 THEN UNTIL ;
7 8 : SET-TIME ( -- ) BEGIN 45 256 * 0
9 CR ." Hour? " INPUT? DROP 256 *
10 CR ." Minute? " INPUT? DROP OR
11 CR ." Second? " INPUT? DROP 256 *
12 CR ." Hit any key to start." CR KEY DROP 33 INTl
13 255 AND IF CR ." Invalid time!" 0 ELSE -1 THEN UNTIL ;
14 15 -->
Screen # 7        crc ver = 16488
0 \ INTERVAL TIME -- itime
1 2 : (ITIME) ( -- time ) GETTIME
3 (U.) TYPE ." :" (U.) TYPE ." :" 
4 (U.) TYPE ." " (U.) TYPE ;
5 6 : ITIME ( -- hr:min:sec:1/100s ) (ITIME) SPACE ;
7 8
9 10
11
12
13
14
15 --> Screen # 8        crc ver = 22508
0 \ CTRL-BREAK SCREEN
1
2 \ The warm start vector is at location 0103H. By jumping to it,
3 \ you can execute the WARM word which initiates a warm start.
4 \ In order to allow the keyboard CTRL/BREAK to use this vector,
5 \ it has to be installed into the DOS interrupt vector table via
6 \ an interrupt routine which can be done by the SETINT word.
7 8 HEX
9 10 LABEL INTRTN STI 20 # AL MOV 20 # AL OUT ( send EOI )
11 0103 #) JMP ( jump to WARM vector ) FORTH
12 : SETINT ( set the interrupt address into interrupt vector )
13 2523 0 0 INTRTN 21 INTO ;
14 15 SETINT ( Execute it ) DECIMAL -->
Screen # 9        crc ver = 65086
0 \ CHECKSUM FOR SCREENS -- 1
1 VARIABLE BADDR VARIABLE CHRCNT
2 : GETBLOCK ( block # -- ) BLOCK BADDR ! -1 CHRCNT ! ;
3 : GETCHR ( -- chr ) BADDR @ CHRCNT @ + C@ ;
4 : DECCNT -1 CHRCNT +! 1 ;
5 : GETNXTCHR 1 CHRCNT +! GETCHR ;
6 : CHKEND CHRCNT 1023 >= ;
7 : ?<>BL BEGIN GETNXTCHR BL = NOT CHKEND OR UNTIL ;
8 9 : ?( GETCHR 40 = ;
10 : ?) BEGIN GETNXTCHR 41 = UNTIL 0 ;
11 : SKIP( GETNXTCHR BL = IF ?) ELSE DECCNT THEN ;
12 13 : ?\ GETCHR 92 = ;
14 : SKIPLINE C/L CHRCNT @ C/L MOD - 1- CHRCNT +! 0 ;
15 : SKIP\ GETNXTCHR BL = IF SKIPLINE ELSE DECCNT THEN ; -->
The first instruction, STI, Set Interrupt flag, is used to be sure that other interrupts can occur when this routine is called. The next instructions, MOV AL, 0 and OUT 20, AL, sends an End Of Interrupt command to the 8259 interrupt controller chip so that other interrupts can be collected by the chip. And, finally, we jump to the warm start vector.

That's the whole interrupt routine. We have to do all these things because, in a normal interrupt sequence, the routine would have to save all registers, and return with an IRET instruction. Since we are not returning from the interrupt routine, certain things must be done — not only those mentioned above, but also the stack has to be cleared. This is done by the warm-start function number of the interrupt that does the checksum calculation. If the character did turn out to be a comment, we have to check to see if we reached the end of the line and then start again on the next character. The checksum is the same as that used in the original version, so the end result will be the same.

There not only are no line delimiters in an F83 screen and spaces, so they won't be counted in the total.

The program was such a good idea, I have been using it ever since the article. When I got F83 however, I found a few new words that messed up the checksum. One of the words used a lot in F83 screens is \. This allows a comment on a line and causes compiling to skip to the end of the line. After trying to modify Suralis' and Brodie's program to compensate for this word and finding, for myself, no easy way of doing it, I decided to rewrite the program to fix the problem.

The way I did it was to count characters. There not only are no line delimiters in an F83 screen, but there are no screen delimiters. So, in order to know where you are in a line when the \ word occurs, you have to be able to count characters in order to get to the end of the line.

To show how I developed the program, take a look at the flow chart for the program which is shown in Figure One. The first thing we do is get the character in the text stream and check to see if it is a blank. If it is, we get the next character. If the character is a ( we have to check to see whether it
comments. The word in the next line, CHKWORD, is used because the space after a word has to be included in the checksum calculation.

The word CHKSUM does all the work in the program and requires an initial value of zero on the stack. It will leave the checksum on the stack when it has finished. I have timed both this version and the older version, and though both take a few seconds to do the calculation, my version is slightly longer. When I get more time, I’d like to speed up the process by doing some of the work in assembly code. The time taken is well worth it, though, since errors found save more time in the long run.

The word CRCCHK is the next step in the process. It requires the block number of the screen to check and leaves the checksum. The final word VER is the same as that used in the old version; in this way you don’t have to change the name if you use this word in your screen-listing words.

Even though this checksum method checks for the two typical comment words, there are two other words in F83 that it doesn’t check for, \(S\) and \(\backslash S\). One indicates a stack comment, and the other jumps to the end of the screen so that numerous lines of comment can be added to the end of a screen. Since I rarely use these words, I felt the time to add them to the program was not worth it to me. If you wish to add them, the method I used in the checksum program makes it not too difficult to do. Let us all know how you did it, if you decide to add them.

I have shown in this article a number of words I have created to do a number of low-level and high-level operations. Especially important and interesting are the ones that get to the BIOS and DOS interrupts. These words allow you to do many new things with Forth that make programming a lot easier.

```
Screen # 10  crc ver = 55006

0 \ CHECKSUM FOR SCREENS -- 2  
1 2 : CALCCRC ( oldcrc chr -- newcrc ) 256 * XOR 8 0 DO DUP 0<
2 IF 16386 XOR DUP + 1+ ELSE DUP + THEN LOOP ;
3 4
5 6 : CHKCHR ?\ IF SKIP\ ELSE ?\ IF SKIP\ ELSE 1 THEN THEN ;
7 : CHKWORD GETNXTCHR BL = IF BL CALCCRC THEN DECCNT DROP ;
8 9 : CHKSUM BEGIN ?<BL CHKEND NOT IF CHKCHR
10 IF GETCHR CALCCRC CHKWORD THEN THEN CHKEND UNTIL ;
11 : CRCCHECK ( block # -- crcvalue ) GETBLOCK O CHKSUM ;
12 13 : VER SCR @ CRCCHECK U. ;
14 15
```

Volume IX, Number 4

Forth Dimensions
NGS FORTH
A FAST FORTH, OPTIMIZED FOR THE IBM PERSONAL COMPUTER AND MS-DOS COMPATIBLES.

STANDARD FEATURES INCLUDE:
- 79 STANDARD
- DIRECT I/O ACCESS
- FULL ACCESS TO MS-DOS FILES AND FUNCTIONS
- ENVIRONMENT SAVE & LOAD
- MULTI-SEGMENTED FOR LARGE APPLICATIONS
- EXTENDED ADDRESSING
- MEMORY ALLOCATION CONFIGURABLE ON-LINE
- AUTO LOAD SCREEN BOOT
- LINE & SCREEN EDITORS
- DECOMPILED AND DEBUGGING AIDS
- 8088 ASSEMBLER
- GRAPHICS & SOUND
- NGS ENHANCEMENTS
- DETAILED MANUAL
- INEXPENSIVE UPGRADS
- NGS USER NEWSLETTER

A COMPLETE FORTH DEVELOPMENT SYSTEM.

PRICES START AT $70

NEW HP-150 & HP-110 VERSIONS AVAILABLE

NEXT GENERATION SYSTEMS
P.O. BOX 2987
SANTA CLARA, CA. 95055
(408) 241-5909

STEP UP TO PROCESSING ARRAYS WITH REVERSE POLISH!

Wayland Software Systems, an innovator in processing arrays with Reverse Polish Notation, announces a new interpreter, vector Rudimentary Language (vecRL). It is now available in pre-release form for people interested in experimenting with this new approach. Eight new vecRL operations under DOS, but eventually it will become a stand-alone DOS compatible operating system. You can own either the source or the object at a very reasonable cost.

VecRL is metacomplied with several arithmetic vocabularies, and more to come. For example, + under the I vocabulary adds untyped 16 bit integers while + under the L vocabulary adds untyped 32 bit long integers. The untyped arithmetic vocabularies, C I L E, provide full FORTH capabilities of 8, 16, 32 bit processing and 16 bit pointers respectively.

Arithmetic vocabularies are under development that process typed array data. For example, the + under the i vocabulary will add 16 bit integer array element by element while + under the l vocabulary adds 32 bit long integer array element by element. The + under the v vocabulary will add typed array element by element using the appropriate addition. These array arithmetic vocabularies will be available in later versions.

Wayland Software Systems offers VecRL because it improves programmer and computer performance. Please call or write for details.

Wayland Software Systems
Advancing the Processing of Arrays with Reverse Polish
10 Shore Drive
Wayland Mass. 01778 USA
(617) 877-9099

(Continued form page 27.)

compiled just as it normally would have been.

Perhaps it seems to be a bit wasteful of memory, to have a field like the one THERE. But I often need just such a field, to be used temporarily. You can use it — with care — instead of the restless PAD field. The patches don’t change the normal behavior of the system, in case of normal compiling or normal interpreting. It is still a Forth system!

However, the method does have some limitations:
1. You cannot interpret structures that nest ?EXEC words. For example:
   IF : TASK ; THEN

   will not work, since PRECOMPILE tries to compile the word : (which is prohibited by ?EXEC).

   2. You must be careful with the code compiled THERE, while it is still running. For example, if you interpret a structure that nests the sequence

   QUERY INTERPRET

   and you try to interpret a ?COMP word by this sequence, it will overwrite the code that already runs THERE, and the system will probably crash.

   Interpret Forth!
NOW FOR IBM PC, XT, AT, PS2 AND TRS-80 MODELS 1, 3, 4, 4P

The Gifted
Computer

1. Buy MMSFORTH before year's end, to let your computer work harder and faster.
2. Then MMS will reward it (and you) with the MMSFORTH GAMES DISK, a $39.95 value which we'll add on at no additional charge!

MMSFORTH is the unusually smooth and complete Forth system with the great support. Many programmers report four to ten times greater productivity with this outstanding system, and MMS provides advanced applications programs in Forth for use by beginners and for custom modifications. Unlike many Forths on the market, MMSFORTH gives you a rich set of the instructions, editing and debugging tools that professional programmers want. The licensed user gets continuos free phone tips and a MMSFORTH Newsletter is available.

The MMSFORTH GAMES DISK includes arcade games (BREAKFORTH, CRASHFORTH and, for TRS-80, FREWAY), board games (OTHELLO and TIC-TAC-FORTH), and a top-notch CRYPTOQUOTE HELPER with a data file of coded messages and the ability to encode your own. All of these come with Forth source code, for a valuable and enjoyable demonstration of Forth programming techniques. Hurry, and the GAMES DISK will be our free gift to you. Our brochure is free, too, and our knowledgeable staff is ready to answer your questions. Write, Better yet, call 617/653-6136.

The Gifted
Computer

(Continued from page 20.)

DM: That's what you have to do to be in the game, I felt.

MH: [to Lori] How did you get into this game?

LC: It started when I took programming in school and liked it. It's rare to find something you really like to do. I don't think I have the entrepreneurial spirit that Rick does, but it's definitely something I like doing.

MH: What about it do you like?

LC: The programming, and — we have a high sense of responsibility, and that comes out. You feel you have a direct effect on what happens.

MH: And that's the part you like — seeing your Machs going out the door?

LC: Yes, and you don't see that in a big company. Rarely do you see it.
FIG CHAPTERS

U.S.A.

• ALABAMA
  Huntsville FIG Chapter
  Tom Konantz (205) 881-6483

• ALASKA
  Kodiak Area Chapter
  Horace Simmons (907) 486-5049

• ARIZONA
  Phoenix Chapter
  4th Thurs., 7:30 p.m.
  Dennis L. Wilson (602) 956-7578
  Tucson Chapter
  2nd & 4th Sun., 2 p.m.
  Flexible Hybrid Systems
  2030 E. Broadway #206
  John C. Mead (602) 323-9763

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

• CALIFORNIA
  Los Angeles Chapter
  4th Sat., 10 a.m.
  Hawthorne Public Library
  12700 S. Grevillea Ave.
  Phillip Wasson (213) 649-1428
  Monterey/Salinas Chapter
  Bud Devins (408) 633-3253
  Orange County Chapter
  4th Wed., 7 p.m.
  Fullerton Savings
  Huntington Beach
  Noshir Jesung (714) 842-3032
  San Diego Chapter
  Thursdays, 12 noon
  Guy Kelly (619) 450-0553
  Sacramento Chapter
  4th Wed., 7 p.m.
  1798-59th St., Room A
  Tom Ghormley (916) 444-7775
  Silicon Valley Chapter
  4th Sat., 10 a.m.
  H-P, Cupertino
  George Shaw (415) 276-5953
  Stockton Chapter
  Doug Dillon (209) 931-2448

• COLORADO
  Denver Chapter
  1st Mon., 7 p.m.
  Steven Sams (303) 477-5955

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

• FLORIDA
  Orlando Chapter
  Every other Wed., 8 p.m.
  Herman B. Gibson (305) 855-4790
  Southeast Florida Chapter
  Coconut Grove area
  John Forsberg (305) 252-0108
  Tampa Bay Chapter
  1st Wed., 7:30 p.m.
  Terry McNay (813) 725-1245

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

• ILLINOIS
  Cache Forth Chapter
  Oak Park
  Clyde W. Phillips, Jr.
  (312) 386-3147
  Central Illinois Chapter
  Urbana
  Sidney Bowhill (217) 333-4150
  Rockwell Chicago Chapter
  Gerard Kusidek (312) 885-8092

• INDIANA
  Central Indiana Chapter
  3rd Sat., 10 a.m.
  John Ogleby (317) 353-3929
  Fort Wayne Chapter
  2nd Tues., 7 p.m.
  IP Univ. Campus, 871 Neff Hall
  Blair MacDermid (219) 749-2042

• IOWA
  Iowa City Chapter
  4th Tues.
  Engineering Bldg., Rm. 2128
  University of Iowa
  Robert Benedict (319) 337-7853

• KANSAS
  Wichita Chapter (FIGPAC)
  3rd Wed., 7 p.m.
  Wilbur E. Walker Co.,
  532 Market
  Arne Flores (316) 267-8852

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

• MICHIGAN
  Detroit/Ann Arbor area
  4th Thurs.
  Tom Chrakiewicz (313) 322-7862

• MINNESOTA
  MNFIG Chapter
  Minneapolis
  Even Month, 1st Mon., 7:30 p.m.
  Odd Month, 1st Sat., 9:30 a.m.
  Vincent Hall, Univ. of MN
  Fred Olson (612) 588-9532

• MISSOURI
  Kansas City Chapter
  4th Tues., 7 p.m.
  Midwest Research Institute
  MAG Conference Center
  Linus Orth (913) 236-9189
  St. Louis Chapter
  1st Tues., 7 p.m.
  Thornhill Branch Library
  Contact Robert Washam
  91 Weis Dr.
  Ellisville, MO 63011

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

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

• NEW YORK
  FIG, New York
  2nd Wed., 7:45 p.m.
  Manhattan
  Ron Martinez (212) 866-1157
  Rochester Chapter
  4th Sat., 1 p.m.
  Monroe Comm. College
  Bldg. 7, Rm. 102
  Frank Lanzafame (716) 235-0168
  Syracuse Chapter
  3rd Wed., 7 p.m.
  Henry J. Fay (315) 446-4600

• NORTH CAROLINA
  Raleigh Chapter
  Frank Bridges (919) 552-1357

• OHIO
  Akron Chapter
  3rd Mon., 7 p.m.
  McDowell Library
  Thomas Franks (216) 336-3167
  Athens Chapter
  Iseal Urieli (614) 594-3731
  Cleveland Chapter
  4th Tues., 7 p.m.
  Chagrin Falls Library
  Gary Bergstrom (216) 247-2492
  Dayton Chapter
  2nd Tues. & 4th Wed., 6:30 p.m.
  CFC. 11 W. Monument Ave.,
  #612
  Gary Ganger (513) 849-1483

• OKLAHOMA
  Central Oklahoma Chapter
  3rd Wed., 7:30 p.m.
  Health Tech. Bldg., OSU Tech.
  Contact Larry Somers
  2410 N.W. 49th
  Oklahoma City, OK 73112

• OREGON
  Greater Oregon Chapter
  Beaverton
  2nd Sat., 1 p.m.
ANNOUNCING

FORTH INTEREST GROUP ROUNDTABLE

NOW AVAILABLE THROUGH

GEInfo
General Electric Network for Information Exchange

* OVER 400 DOWNLOADABLE FILES OF FORTH INFORMATION & CODE

* ON-LINE REAL TIME CONFERENCING

* E-MAIL CONTACT WITH OTHER FIG MEMBERS

SPECIAL SIGN-UP FOR FIG MEMBERS ONLY

$18.00

Includes GEInfo Manual plus 3 FREE Hours on GEInfo

Using Your Modem Call 1-800-638-8369, Type "HHH" (CR)
Following the U# prompt, type "XJM11849,GENIE" (CR)

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