

# FORTH DIMENSIONS

OCTOBER/NOVEMBER 1978

VOLUME 1, NO. 3

## CONTENTS

<b>HISTORICAL PERSPECTIVE</b>	<b>Page 24</b>
<b>CONTRIBUTED MATERIAL</b>	<b>Page 24</b>
<b>DTC vs ITC for FORTH</b> David J. Sirag	<b>Page 25</b>
<b>D-CHARTS</b> Kim Harris	<b>Page 30</b>
<b>FORTH vs ASSEMBLY</b> Richard B. Main	<b>Page 33</b>
<b>HIGH SPEED DISC COPY</b> Richard B. Main	<b>Page 34</b>
<b>SUBSCRIPTION OPPORTUNITY</b>	<b>Page 35</b>

## HISTORICAL PERSPECTIVE

FORTH was created by Mr. Charles H. Moore in about 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of his dissatisfaction with available programming tools, especially for automation. Distribution of his work to other observatories has made FORTH the de-facto standard language for observatory automation.

Mr. Moore and several associates formed Forth Inc. in 1973 for the purpose of licensing and support of the FORTH Operating System and Programming Language, and to supply application programming to meet customers' unique requirements.

FORTH enjoys a synergism of its features. It has none of the elephantine characteristics of PL/1 or FORTRAN. It has a density and speed far surpassing BASIC, but retains an interactive nature during program development. Since it is extensible, special words are easily defined

to give it the terseness of APL. Its clarity and consistency result from being the product of a single mind. (as were APL and PASCAL).

Although the language specification and many implementations are in the public domain, many other implementations and application packages are available as program products of commercial suppliers.

The FORTH Interest Group is centered in Northern California. It was formed in 1978 by local FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications. About 300 members are presently associated into a loose national organization. ('Loose' means that no budget exists to support any formal effort.) All effort is on a volunteer basis and the group is associated with no vendors.

;S W.F.R 8/20/78

---

## CONTRIBUTED MATERIAL

FORTH Interest Groups needs the following material :

1. Technical material for inclusion in FORTH DIMENSIONS. Both expositions on internal features of FORTH and application programs are appreciated.
2. Name and address of FORTH Implementations for inclusion in our publications. Include computer requirements, documentation and cost.
3. Manuals available for distribution. We can purchase copies and distribute, or print from your authorized original.
4. Letters of general interest for publication in this newsletter.
5. Users who may be referenced for local demonstration to newcomers.

## DTC VERSUS ITC FOR FORTH ON THE PDP-11

By David J. Sirag  
Laboratory Software Systems, Inc.  
3634 Mandeville Canyon Road, Los Angeles, CA 90049

During the design of LABFORTH, the FORTH implementation by Laboratory Software Systems, the choice had to be made between direct threaded code (DTC) and indirect threaded code (ITC). A detailed analysis showed DTC to be significantly superior to ITC in both speed and size. This analysis contradicts the findings of Dewar (ACM June 1975) which were referenced in the "Threaded Code" article in the August 1978 issue of FORTH Dimensions. Dewar compared his use of ITC with DTC as used for PDP-11 FORTRAN. His analysis does not apply to the implementation of FORTH on the PDP-11.

The FORTH analysis involves 3 types of definitions - low level (CODE), high level (COLON), and storage (variable, etc). The low level definitions will be encountered most frequently by far because of the pyramidal nature of FORTH definitions. On the other hand, storage definitions will be encountered far less frequently in FORTH than in FORTRAN because in FORTH the stack is used extensively while in FORTRAN no stack is available. Also, when storage locations are used in FORTH operators are available which minimize the number of references. For example, in FORTRAN

```
COUNT = COUNT + 1
```

involves 2 references to the variable COUNT, while in FORTH

```
COUNT 1+!
```

involves only 1 reference. It should be noted that in LABFORTH, 1+! is a primitive, but it is not in some other versions of FORTH. Another factor which reduces the references to storage locations is that in FORTH literals are placed in line and handled by a reference to the LITERAL (low level) routine.

The DTC and ITC routines for the 3 types of definitions are shown below, they are condensed to show only the relative PDP-11/40 overhead. The register notation in the routines is as follows:

Q is the cue register (R5) which points to the next address. It is called IC (instruction counter) in some literature.

S is the stack pointer (R4).

R is the return stack pointer (R6).

P is the program counter (R7).

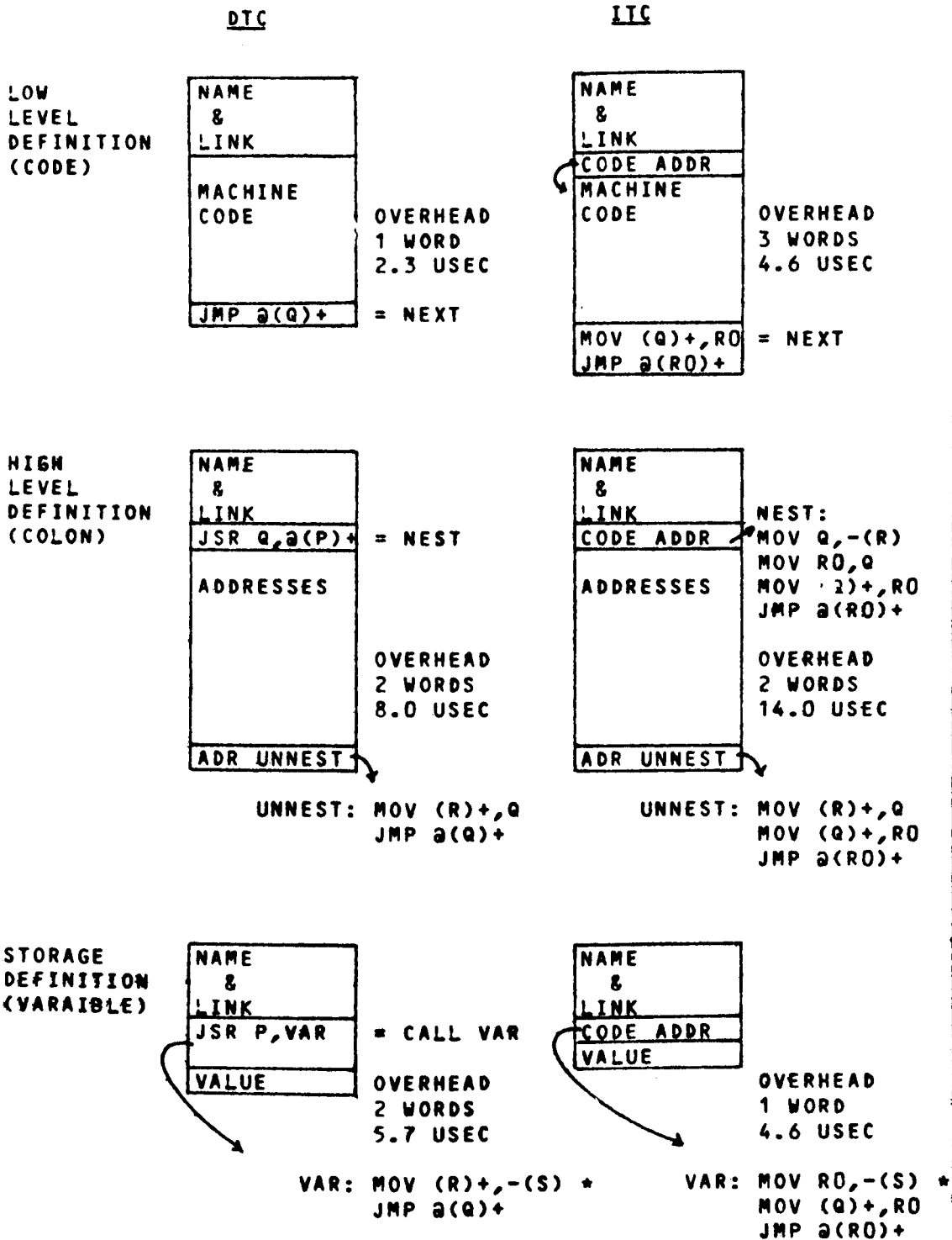
RD is a temporary register assumed to be available.

-->

PAGE 25

FORTH INTEREST GROUP ..... P.O. Box 1105 ..... San Carlos, Ca. 94070

DTC AND ITC ROUTINES



\* The push instruction itself is not counted in the overhead

The distinction between DTC and ITC as applied to FORTH is that in DTC executable machine code is expected as the first word after the definition name; while, in ITC the address of the machine code is expected. Thus the DTC space advantage in the entry to a low level definition is obvious. The machine code of the low level definition terminates with the "NEXT" routine. In DTC NEXT is a 1 word routine while in ITC the extra level of indirection results in a 2 word routine (Note: a JMP NEXT would also take 2 words).

In the high level definition the machine code of the "NEST" routine is stored in line for DTC, but since it is only 1 word, it takes no more room than the pointer to the "NEST" routine. However, the 1 instruction for DTC takes considerably less time to execute than the 4 instructions for ITC (Note: replacing the last 2 instructions with JMP NEXT would take even more time). The remaining words in the high level definition are addresses in both cases. The last address points to the UNNEST routine which again is more complex for ITC because of the additional indirection.

In the storage definition case the machine code of the subroutine call to the appropriate processor (VAR in the example) is stored in line. This requires 2 words not including the the storage for the variable itself. The storage words follow the call and can be thought to be the parameters for the call. Thus in this case, the 1 word code address for ITC represents a 1 word advantage over the subroutine call. The execution time is also slightly in favor of ITC, even though 3 instructions are executed in both cases.

DTC VERSUS ITC OVERHEAD SUMMARY			
	<u>DTC</u>	<u>ITC</u>	<u>DTC ADVANTAGE</u>
Low level (CODE)	1 word 2.3 usec	3 words 4.6 usec	2 words 2.3 usec
High level (COLON)	2 words 8.0 usec	2 words 14.0 usec	0 words 6.0 usec
Storage (VARIABLE)	2 words 5.7 usec	1 word 4.6 usec	-1 word -1.1 usec

The summary table shows that DTC has the overhead advantage in both low level and high level definitions; while ITC has the advantage in storage definitions. Considering the high occurrence of low level definitions and the low usage of storage definitions, one can see that a FORTH implementation with DTC has a significant speed and space

-->

advantage over one using ITC. To make the advantage more concrete weights should be assigned to the various definition types. If we have a program containing 500 definitions (including the standard FORTH definitions), we might expect 200 low level, 250 high level, and 50 storage definitions. Using these numbers the size advantage of low level, high level, and storage should be weighted .4, .5, and .1 respectively. During the execution of a program, we might expect the frequency of occurrence of low level, high level, and storage to be 60%, 20%, and 20% respectively. The result of applying these weights is shown in the following table.

WEIGHTED ADVANTAGE OF DTC OVER ITC		
	<u>SIZE ADVANTAGE</u>	<u>SPEED ADVANTAGE</u>
Low level	2 x .4 = .8 words	2.3 x .6 = 1.38 usec
High level	0 x .5 = 0 words	6.0 x .2 = 1.2 usec
Storage	-1 x .1 = -.1 words	-1.1 x .2 = -.22 usec
Weighted advantage	<u>.7 words</u>	<u>2.4 usec</u>

Thus using the weighted advantage for DTC we would expect to save .7 words in each of the 500 definitions which is a total of 350 words. Also each time a definition is executed the overhead would be 2.4 usec less. This may represent a savings of 20 or 30% of the total execution time of the frequently used short definitions.

The remaining advantage that is claimed for ITC is one of machine independence because no machine code appears in the code generated by the compiler. But even this advantage is illusory since FORTH programs are transported in source form. In fact on most systems they are compiled each time they are loaded via the LOAD command. Thus, after a FORTH system is hosted on a given computer, the machine code that is generated by the compiler is suitable for that particular machine; this includes the machine code generated for the DTC routines. If one did try to introduce the concept of FORTH portability at the object code level by restricting the programs to high level definitions and placing all machine code in a run-time package, he would still probably have machine dependencies in byte versus word addresses, floating point format, and character string representation. In any case, current FORTH implementations do not claim transportability at the object code level. --->

The analysis of DTC versus ITC has shown that when the special situation presented by FORTH on the PDP-11 as opposed to FORTRAN is considered, use of DTC provides significant advantages over ITC in both speed and size. Thus LABFORTH was implemented using DTC. However, if it is rehosted on another computer, the choice may be different. The change would be handled as part of the rehosting effort along with all the other changes which would be required.

;S DJS

FORTH Interest Group  
787 Old Country Road  
San Carlos, CA 94070

LABORATORY SOFTWARE SYSTEMS, INC.  
3634 MANDEVILLE CANYON ROAD  
LOS ANGELES, CALIF. 90049  
(213) 472-6995

Dear Figgy,

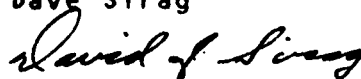
FORTH Dimensions is just the sort of communications vehicle which is needed by the FORTH community for both users and vendors. My payment for a subscription is enclosed.

As Dr. R.M. Harper indicated in an earlier letter, we at Laboratory Software Systems have developed a version of FORTH on the PDP-11 called LABFORTH. As the name implies, LABFORTH contains features which make it particularly suitable for the scientific laboratory environment. This environment includes high speed data collection and analysis; thus particular attention is given to making LABFORTH fast. For this reason the direct versus indirect threaded code discussion in the Thread Code article in the August/September 1978 issue of FORTH Dimensions was of particular interest. Our analysis of DTC versus ITC was an important aspect of the effort to design LABFORTH for maximum speed. DTC proved to be faster than ITC and as a bonus required less space. An article on this analysis is enclosed for your paper. It contradicts Dewar's analysis of DTC versus ITC for DEC's FORTRAN, but his analysis cannot really be applied to FORTH. If DEC had used DTC in a more elegant manner, DTC may also have fared better in the FORTRAN case.

Hopefully the DTC advantages will persuade you to delete the requirement that FORTH be implemented with ITC. The programming techniques used in implementing FORTH ought to be left to the designer and his results should to be evaluated by benchmarks.

I look forward to your next issue of FORTH Dimensions.

Dave Sirag



Laboratory Software Systems, Inc.

PAGE 29

FORTH INTEREST GROUP ..... P.O. Box 1105 ..... San Carlos, Ca. 94070

D-CHARTS

Kim Harris

An alternative style of flowcharts called D-charts will be described. But first the purpose of flowcharting will be discussed as well as the shortcomings of traditional flowcharting.

A flowchart should be a tool for the design and analysis of sequential procedures which make the control flow of a procedure clear. With FORTH and other modern languages, flowcharts should be optimized for the top-down design of structured programs and should help the understanding and debugging of existing ones. An analogy may be made with a road map. This graphic representation of data makes it easy to choose an optimum route to some destination, but when driving, a sequential list of instructions is easier to use (e.g., turn right on 3rd street, left on Ave. F, go 3 blocks, etc.). Indentation of source statements to show control structures is helpful and is recommended, but a two dimensional graphic display of those control structures can be superior. A good flowchart notation should be easy to learn, convenient to use (e.g., good legibility with free-hand drawn charts), compact (minimizing off-page lines), adaptable to specialized notations, language, and personal style, and modifiable with minimum redrawing of unchanged sections.

Traditional flowcharting using ANSI standard symbols has been so unsuccessful at meeting these goals that "flowchart" has become a dirty word. This style is not structured, is at a lower level than any higher level language (e.g., no loop symbol), requires the use of symbol templates for legibility, and forces program statements to be crammed inside these symbols like captions in a cartoon.

D-charts have a simplicity and power similar to FORTH. They are the invention of Prof. Edsger W. Dijkstra, a champion of top-down design, structured programming, and clear, concise notation. They form a context-free language. D-charts are denser than ANSI flowcharts usually allowing twice as much program to be displayed per page. There are only two symbols in the basic language; however, like FORTH, extensions may be added for convenience.

Sequential statements are written in free form, one below the other, and without boxes.

```
statement
next statement
next statement
:
```

The only "lines" in D-charts are used to show nonsequential control paths (e.g., conditional branches, loops). In a proper D-chart, no lines go up; all lines either go down or sideways. Any need for lines directed up can be (and should be) met with the loop symbols. This simplifies the reading of a D-chart since it always starts at the top of a page and ends at the bottom.

It is customary to underline the entry name (or FORTH definition name) at the top of a D-chart.

2-WAY BRANCH SYMBOL

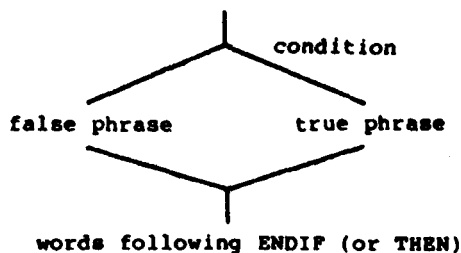
In FORTH, this structure takes the form:

```
condition IF true phrase
                ELSE false phrase
                THEN .
```

Another FORTH structure which is used for conditional compilation has more mnemonic names:

```
condition IFTRUE true phrase
                OTHERWISE false phrase
                ENDIF .
```

The D-chart symbol has parts for each of these elements:

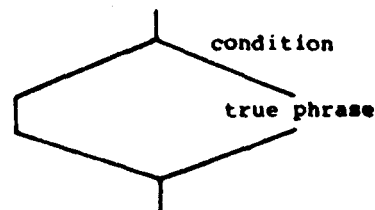


The "condition" is evaluated. If it is true, the "true phrase" is executed; otherwise, the "false phrase" is executed. The words following ENDIF (or THEN) are unconditionally executed.

If either phrase is omitted, as with

```
condition IF true phrase THEN
```

a vertical line is drawn as shown:

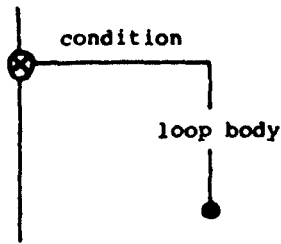


---



LOOP SYMBOL

The basic loop defining symbol for D-charts is properly structured.



The switch symbol:



indicates that when the switch is encountered, the "condition" (on the side line) is evaluated.

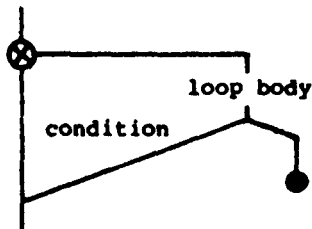
1. If the "condition" is true, then the side line path is taken; if false, then the down line is taken (and the loop is terminated).
2. If the side line is taken, all statements down to the dot are executed. The dot is the loop end symbol and indicates that control is returned to the switch.
3. The "condition" is again evaluated. Its outcome might have changed during the execution of the loop statement.

Repeat these steps starting with Step 1.

This symbol tests the loop condition before executing the loop body. However, other loops test the condition at the end of the loop body (e.g., DO .. LOOP and BEGIN .. END) or in the middle of the loop body. This loop symbol may be extended for these other cases by adding a test within the loop body. Consider the FORTH loop structure

BEGIN loop body condition END .

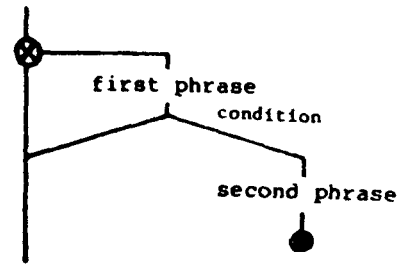
The loop body is always executed once, and is repeated as long as condition is false. The D-chart symbol for this structure would be:



A more general case is

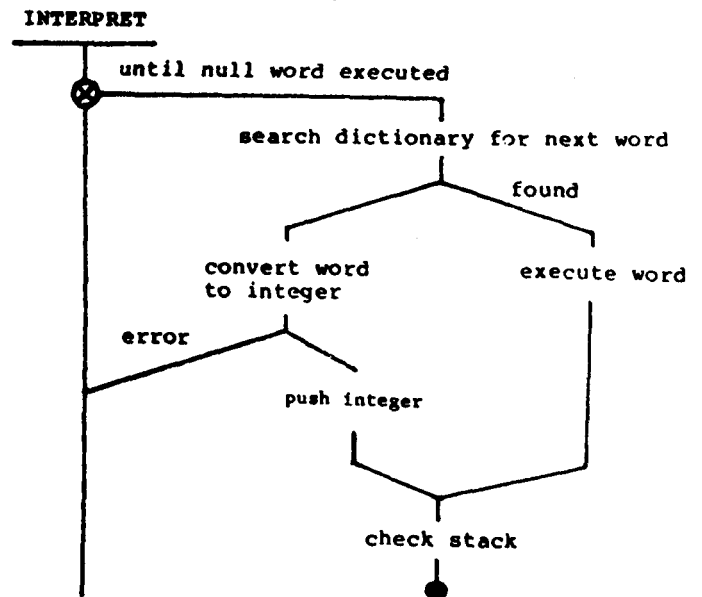
BEGIN first phrase  
condition IF second phrase  
AGAIN

which is explained better graphically than verbally:



Both previous symbols may be properly nested indefinitely. The following example shows how these symbols may be combined. This is the FORTH interpreter from the P.I.G. model.

```
: INTERPRET BEGIN ( ' ) IF HERE NUMBER
ELSE EXECUTE
THEN
?STACK
AGAIN ;
```



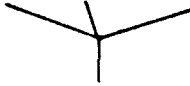
-->

### n-WAY BRANCH SYMBOL

A structured n-way branch symbol (sometimes called a CASE statement) may be defined for convenience. (It is functionally equivalent to n nested 2-way branches). One style for this symbol is:



first case    second case    ..    last case



The condition is usually an index which selects one of the cases. The rejoining of control to a single line after the cases are required by structured programming. Depending on the complexity of the cases, this symbol may be drawn differently.

D-charts are efficient and useful. They are vastly superior to traditional flowchart style.

;S KIM HARRIS

P.O. Box 8045  
Austin, TX 78712  
November 3, 1978

Editor, Forth Dimensions:

### SYSTEM LANGUAGE 1

Thank you for your card and subsequent letter. I am sorry that I did not get back to you sooner with a copy of the source code for my FORTH system. Frankly, I was surprised that you are interested in the system, since it is rather limited in facilities and conforms with no other FORTH version in terms of names. I stopped work on the system just about the time I began to receive manuals from DECUS and the 6502 FORTH form FIG. I can see now how I would add an assembler, text editor, and random block i/o to the system, but my duties at work and at school preclude any further development of U.T. FORTH for now.

I want to especially thank you for informing me of Paul Bartholdi's visit to the University of Texas. I was able to meet with him and we had a very stimulating discussion for about an hour and a half. I was surprised to learn from him how widely FORTH is used commercially, though usually under other names. We also discussed two extensions to the language that I believe greatly enhance it: (1) syntax checking on compilation for properly balanced BEGIN..END and IF..ELSE..THEN constructs, and (2) the functions "n PARAMETERS" and "PAR|" to "PARV" that allow explicit reference to parameters on the stack. Finally, he showed me some programming examples from the FORTH manual he wrote which provide first-hand proof of the ease of programming rather sophisticated problems in FORTH. It is especially important because most people in the computer science department here respond to my presentation of FORTH with a resounding lack of interest. After all, they keep abreast of the field and if they have not heard of it....

I have been promoting FORTH among the local computer clubs and look forward to the results of FIG's micro computer efforts. Please keep in touch.

Sincerely yours,  
Greg Walker

SL/1 was written by Emperical Research Group, Inc. to be exactly what it says it is, a SYSTEM language. SL/1 is a small interactive incremental compiler that generates indirect threaded code. It is a 16 bit pseudo machine for use on mini and micro computers. New definitions can be added to an already rich set of intrinsic instructions. It is this extensibility that allows any user to create the most optimum vocabulary for his individual application.

SL/1 is a virtual stack processor. Using the RPN concept for both variables and instructions makes it possible to extend stepwise programming to include stepwise debugging. SL/1 does this quite nicely. The RPN stack is also one of the most effective means of implementing top down design, bottom up coding.

SL/1 operates on a principle of threaded code. All of the elements of SL/1 (procedures, variable, compiler directives, etc.) reference the previous entry. Thus, each code indirectly "threads" the others and is in turn threaded by the code following it. Because SL/1 is a pseudo machine, portability between different processors and hardware is readily accomplished. The low level interpreter is really the P-machine. It is small (only 11 bytes are used), and fast.

One of the most powerful features of SL/1 is the fact that it uses all on-line storage media as virtual memory. In effect the user can write programs in SL/1 using the full capacity of disk storage and never be concerned with placement of information on the disk. SL/1 allows you to program machine code procedures in assembler using a high level language. This can optimize I/O or math routines.

The above information was excerpted from a press release of November 3, 1978. For further information, contact Mr. Dick Jones, Emperical Research Group, Inc., 28206 144th Avenue, S.E., Kent, WA 98031. Phone (206) 631-4851.

FORTH VS. ASSEMBLY  
By Richard B. Main  
Neptune UES, Pleasanton, CA

Here are some facts regarding Forth object size and execution speeds versus Assembly coding.

Forth, Inc., some programmers (myself included), and others have made some pretty incredible statements about Forth code resulting in less memory required (!) and execution speeds as fast as Assembly written code (!!). To help clear the air I'll try to explain those two outrageous claims.

First, Forth code can run as fast, but not faster, using a constructional statement called "Code" which is followed by a sort of mnemonic machine code string and a jump back to the Forth inner interpreter. It isn't reasonable to just have one big code statement for the whole program. So this gets us into another Forth constructional statement called a "colon definition".

Colon statements cost speed but save program memory over Assembly. Colon statements constitute the "high level" aspect of Forth but let's get back to the point.

An example "code" statement in Forth to handle the character input from a CRT to an Intel SBC 80/20 would be:

```
CODE KEY BEGIN ED INP RRC RRC CS END
          EC INP A L MOV 0 H MVI HPUSH JMP
```

NOTE: Forth code statements allow begin-end and if-else-then constructs within the assembly. Also Forth requires source-destination-operand organization of each assembly statement (A L MOV instead of MOV L,A).

This exact same routine in Assembly language would be:

```
          ORG $           ;place in next avail
KEY:
BEGIN: IN EDH           ;input CRT status
        RRC             ;rotate receiver ready
        RRC             ;into carry bit
END:   JNC BEGIN
        IN ECH          ;input CRT data
        MOV L,A         ;push data on
        MVI H,0         ;stack in 16-bit
        JMP HPUSH      ;format
```

By entering ' KEY 0D DUMP on the Forth system you'll get the object code displayed as:

```
4000 DB 0F 0F D2 00 40 DB EC 6F
      26 00 C3 41 00
```

This is exactly what the Assembly code would produce if ORG'ed at 4000H and the label HPUSH was at 41H.

Reviewing the example Forth code statement: "BEGIN" produced no object but simply acted as a label for "END" and provided the JNC address for END. "CS" simply provided the JMP type for END, in this case JNC. "CS NOT END" would have complemented the jump type and produced JC.

The above examples while not especially exciting on the surface are quite interesting when you're actually writing these programs on a system installed with Forth and one that isn't. Using standard disk-based Assembler system you'd probably have to open an edit file, write the program, close the edit file, call the assembler, and load the object file so you could use the debug program to execute. Maybe 10-30 minutes depending on the problems you have along the way. In Forth, you'd enter the code statement on the command line, carriage return, type "KEY", (CR), and it's executing. 30 seconds maximum! If you liked the way "KEY" executed you'd save it off on the disk using the Forth Editor. (Another 20 seconds.)

The colon statement in Forth was said to save room in memory over Assembly, and provide the high level language ability. An example code statement that would read the CRT keyboard command messages and then execute the desired action could look like:

```
: KEYBOARD 64 0 DO KEY 7F AND DUP 0D =
          IF LEAVE THEN LOOP EXECUTE ;
```

Keyboard is the label of this routine. Every other word (DO, KEY, AND, =, LEAVE, THEN, LOOP, and EXECUTE) requires two bytes of memory. 8-bit numbers require 3 bytes, 1 for the number and 2 for a routine that differentiates numbers from words and provides these numbers on the stack for use by succeeding operations, e.g., 64 and 0 for 'DO'.

The memory saving can be visualized by thinking of the routine "keyboard" as a routine that looks like:

```
KEYBOARD: CALL 64      ;a program
          CALL 0       ;a program
          CALL DO      ;to start a 64 loop
          CALL KEY     ;to input data
          CALL 7F      ;# for AND
          CALL AND     ;to AND it
          CALL DUP     ;to DUP data
          CALL 0D      ;for (CR) test
          CALL IF      ;for (CR) test
          CALL LEAVE   ;if (CR) leave loop
          CALL THEN    ;to complete IF
          CALL LOOP    ;to loop 64 times
          CALL EXECUTE ;to DO command
          CALL ";"     ;to DO next one
```

Looking at it this way, each CALL takes a byte. Fourteen bytes could be saved if the CALL OP CODE could be eliminated. The result would be the two byte address' of everything to CALL. The innermost Forth interpreter uses these address' in sequence and is about 12 bytes of memory code and has the label "NEXT".

Thus, for just this single example, 14 bytes were saved, at the cost of 12 bytes for "NEXT". But every colon and code statement used "NEXT" so the memory savings build because "NEXT" is executed so many times. The justification in using sub-routine calls in Assembly code versus inline code is based on how many times it is called. "NEXT" is completely justified because it is called an enormous number of times. Forth, Inc., has stated "NEXT" would be an

-->

excellent micro-code to be included in a CPU OPCODE set and I'd have to agree. Before a "NEXT" OPCODE would be implemented in MOS processor-like 8085, 6800, or the like, Forth is going to have to become quite dear to the industry. So I don't see it happening except in some 2900 bit-slice implementations.

All this concern about micro-coding "NEXT" has its root. "NEXT" is executed between each word in a colon statement and between each word of a word that itself is the name of a colon statement. Therefore, "NEXT" slows things down during execution, but is redeeming since it saves space and allows the high level nature of Forth.

To keep things moving quickly in the execution of Forth programs, colon statements should contain a few words defining the action of the defined colon statement and each word should be very closely connected to a code statement as possible (since code statements run at full machine speed). Also, each word in a colon statement should be powerful, if the word is the label of a code statement, this could mean large code statements.

Large code statements can quickly get out of hand with more than two lines (line in the example of "KEY"), because of the lesser ability to comment each OPCODE as in Assembly. So Forth, Inc., has stated code statements should be kept short and sweet. It's really up to the user to trade off readability for speed.

The naming of colon and code statement labels can really improve readability if you put some thought into the naming.

As was said earlier, the Forth program statement can be executed by entering it on the command line, then typing the name for execution. Colon statements are included in this ability and extremely fast coding and debugging is the result.

I really object to paying \$2,500 for any software, but Forth is worth it. (They'd probably sell more if it wasn't so expensive.) Besides the price there seems to be a few other impediments to Forth gaining a more rapid popularity growth. (1) It does take some getting used to. (2) There's not many Forth systems and programmers around. (3) People, in my judgment, are too quick to condemn it.

;s REM

HIGH SPEED DISK COPY  
By Richard B. Main  
Neptune UES, Pleasanton, CA

To really get fast disk copies on your MDS-800 (TM Intel Corp.) Forth systems, add this program to your diskling load:

```

0 ( HIGH SPEED DISK COPY RBM-781001 )
1 16384 CONSTANT SCRATCH
2 2000 CONSTANT BIAS
3 26 CONSTANT TRACK
4 4 CONSTANT READ
5 6 CONSTANT WRITE
6 26 CONSTANT ALL
7 : DUPLICATE FMT 77 0
8 DO SCRATCH I TRACK *
9 READ ALL I/O I .
10 SCRATCH I TRACK *
11 BIAS + WRITE ALL I/O
12 STATUS IF [ ERROR] LEAVE THEN
13 LOOP FLUSH CR [ COPY ] 7 ECHO ;
14 ;S DUPLICATE TAKES 80 SECONDS TO
15 FORMAT AND COPY ENTIRE NEW DISK.
```

The main reason this program will take only 80 seconds to make a copy is whole tracks are read from the master disk in drive # and whole tracks are written to the copy in drive 1. But, alas, you'll need 3328 bytes of continuous RAM to run this

program. The constant named SCRATCH provides the first address of the 3328 RAM bytes needed.

DUPLICATE when executed calls FMT to format the disk in drive 1. "77 # DO" sets up a DO-LOOP to copy all 77 tracks. I/O requires SCRATCH (location) I (the track and index of the loop) TRACK \* (to compute block # for I/O) READ (from drive #) and ALL (for # of sectors). I/O will perform the disk operation. "I" prints the current track being copied to entertain the operator. Next, SCRATCH again gives the scratch area for I/O and I TRACK \* BIAS + provides the equivalent block number in drive 1 for I/O.

WRITE ALL instructs I/O to write all 26 sectors from scratch area. I/O performs the disk operation. STATUS pops the disk status byte from location 2#H and if non-zero prints ERROR and leaves the loop. Else the loop repeats and FLUSH is executed for the heck-of-it. COPY is printed and BELL is echoed to CRT to signal completion.

;s REM Oct. 1978