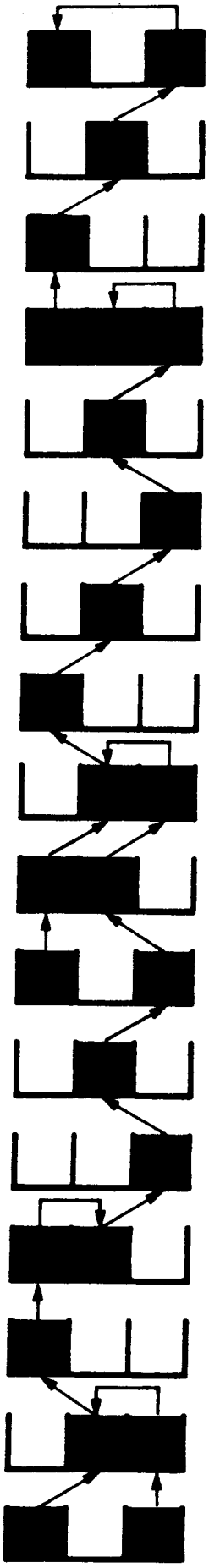


FORTH DIMENSIONS

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

VOLUME I

Numbers 1 - 6

I/1-6/500

FORTH DIMENSIONS INDEX
VOLUME I,II,III

INDIVIDUAL WORDS DEFINED

MISC. ENTRIES

SCREENS OF FORTH CODE

INDEX COMPILED COURTESY OF
M. TASSANO
936 DELAWARE WAY
LIVERMORE,CA. 94550

FORTH DIMENSIONS

JUNE/JULY 1978 VOLUME 1 NO. 1

EDITORIAL:
WHAT IS THE FORTH INTEREST GROUP?

The Forth Interest Group, which developed in the fertile ground of the computer clubs of the San Francisco Bay Area, grew in a few months from nothing to where we are now getting several letters a day from all over the country. With this increasing public interest we need to let people know what we are doing and why, what we would like to see happen, how others can be involved, and what we can and cannot do.

We are involved because we believe that this language can have a major effect on the usefulness of computers, especially small computers, and we want to see it put to the test. Increasingly software is becoming the critical, limiting factor in the computer industry. Large software projects are especially difficult to develop and modify. Few are happy with prevailing operating systems, which are huge, hard to understand, incompatible with each other, and without unity of design.

The Forth language is its own operating system and text editor. It is interactive, extensible (including user-defined data types), structured, and recursive. Code is so compact that the entire system (mostly written in Forth) usually fits in 6K bytes, running stand-alone with no other software required, or as a task in a conventional operating system. One person can understand the entire Forth system, change any part of it, or even write a new version from scratch. Run-time efficiencies are as little as 30% slower than straight machine code, and even less if the system's built-in assembler is used. When the assembler is not used, programs can be almost completely transportable between machines. Any large Forth program is really a special-purpose, application-oriented language, greatly facilitating maintenance and modification. We don't yet have conclusive data, but typical program development times and costs seem to be a fraction of those required by Fortran or assembly. Forth is especially useful for real-time, control-type applications, for large projects, and for small machines.

The problem is availability. Users have shown an ease of learning after they have a system available. The Forth characteristics of postfix notation, structured conditionals, and data stacks are best understood by use. To encourage Forth programmers, we need readily available systems even of modest performance. We hope that three levels will be available:

1. Demonstration - free (or under $20.) introductory version without file structure which compiles and executes from keyboard input.

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

2. Personal - low cost ($10. to $100.) with RAM or tape based files.

3. Professional - Commercial products for lab or industrial use and software development. ($1000. to $2500.)

Today the serious personal computer user holds the key to wider availability of the language. These users - generally engineers, businessmen, programmers - combine professional competence and commitment with the freedom to try new methods which may require a lot of time and tinkering with no definite guarantee of payoff. Practically everyone involved with the Forth Interest Group has both a personal and a professional interest in computers.

The Forth Interest Group is non-profit and non-commercial. We aren't associated with any vendor, no one is making money from it, and we are all busy with other work. We are an information clearinghouse and want to encourage distribution of all three of the previously mentioned levels of Forth. We do not have a Forth system for distribution at this time, and we don't want to get into the software or mail-order business because this is best left to companies or individuals committed to that goal. Naturally our critical issue is how to keep going over the long haul with volunteer energy. We need cost-effective means of information exchange.

At present we are writing for professional media, putting out this simple newsletter, and holding occasional meetings in the Bay Area. Also, we are developing a major technical and implementation manual, to be published in a journal form as four installments, available by subscription. While we cannot answer all of the mail individually, we certainly read it all, to answer it in the newsletter. While we cannot fill orders for software or literature, we will try and point you to where it is available. We welcome your input of information or suggestions, how you could help, what you would like to see happen, and where we should go from here.

Dave Bengel - Dave Boulton - Kim Harris - John James
Tom Olsen - Bill Ragsdale - Dave Wyland

EVOLUTION OF A F.I.G. FORTH FREAK

By Tom Olsen

I have been actively involved in the personal computing movement since early in 1974 when I shelled out $120. for an 8008 chip. Since that time my hardware and software have evolved into a very powerful and useful system, of which FORTH is a principal component. The system consists of an LSI-11, 28K of memory, 2 Diablo disks, an LA30 DECWRITER, a Diablo HYTYPE-I printer, a VDM-1 display, and dual floppy disk drives. Obtaining an operating system which would effectively utilize all of this hardware initially appeared to be much too expensive for an individual to buy, and far too complex to write from scratch. This attitude changed when early in 1976 I read a technical manual describing the internal organization of a relatively unknown "language" called FORTH. Here was a programming system which included not only an editor, assembler, and file management system, but the inherent capability to be rapidly expanded to perform any computer function I could define. The best part was the fact that the central core of this programming system was relatively small and would easily fit into 3K of memory. The large majority of the system programming could be done in terms of high level functions which I would have the freedom to define.

After about three months of late nights and pulling my hair out, I finally had a stand alone FORTH system which I could bootstrap and then use to load application vocabularies from disk. Once the basic implementation was fully debugged my general throughput of useful application software increased to a level I never would have thought possible. I can't over-emphasize the satisfaction associated with implementing the language from scratch. An added benefit of this approach is the flexibility derived by having a 100% understanding of ALL of the code your machine is executing.

Today I have application vocabularies which can do everything from playing a BACH minuet on a computer controlled synthesizer to generating, sorting, and printing the FORTH INTEREST GROUP mailing list. It is my hope that with the continued growth of the FORTH INTEREST GROUP and the establishment of some syntactical standards, widespread exchange of applications vocabularies will greatly enhance the computing power of all users of FORTH-like languages.

USING FORTH FOR TRADEOFFS BETWEEN HARDWARE/FIRMWARE/SOFTWARE

By Dave Wyland

FORTH provides a unique capability for changing the tradeoffs between software, firmware, and hardware. This capability derives from the method of nested definition of operators in FORTH.

Firmware (i.e. microprogramming) can add new instructions to the instruction set of an existing machine. New hardware designs can also add instructions to an existing set, with the Z80 upgrade of the 8080 serving as a good example. These new instructions could significantly improve the throughput of a system in many cases: however these new instructions cannot be used with existing software without rewriting the software. This is because current software methods such as assembler language, BASIC interpreters, FORTRAN compilers, etc., create software programs as one list of instructions. To add improved instructions to a program generated by a FORTRAN compiler involves rewriting the compiler to efficiently use the new instructions and then recompiling the program. This is not a trivial task since decisions must typically be made as to when to use a new instruction. This is why each new machine requires a new, rewritten FORTRAN compiler. The fact that the job has been done many times before is not very comforting.

With FORTH, however, operations are built-up of nested definitions with a common functional interface between operations. If a new instruction has been added to the computer's instruction set, it can be added to the FORTH system by changing a small number of definitions, in the typical case. The method can be quite straightforward. Each new instruction does an operation which would have required several instructions in the previous case. By identifying those FORTH definitions which could benefit from the new instructions and recoding them to include it, all software which uses the modified definitions is immediately improved. Also, very few definitions will have to be modified: only those elementry definitions with a high frequency of use need be modified to achieve throughput increase near the maximum possible.

Since the FORTH definitions are nested, the throughput gain of new instructions can be achieved by modifying a small amount of code, and the remaining structures remain unchanged. All existing application programs, translators, etc. are immediately improved without change!

Note that this process is reversable. Programs created for an existing system which has an extensive instruction set can be run on a simpler machine by converting the appropriate code based definitions to nested (colon) definitions. The process is also incremental: new instructions can be added one-at-a-time as desired.

FORTH LEARNS GERMAN

By John James

Forth now understands German, thanks to the following redefinitions of the operations in the Decus (Caltech) Forth manual.

Many of the operations were mathematical symbols, and of course they did not have to be translated. Of the rest, the control operations (IF, THEN, ELSE, BEGIN, END, DO, LOOP, +LOOP) are special, because they are "immediate operations"; that is, they are executed at compile time. Just redefining their names would not work, because they would try to execute right in the definitions. So their original definitions were copied, but with the German names.

Program development time for bilingual capability, two hours. Memory required, 600 bytes. Effect on run-time execution speed, zero.

```
BLOCK 30
 1 ( GERMAN.  JJ, 6/19/78)
 2 : ABWERFEN DROP ;     : UBER OVER ;
 3 : VERT SWAP ;    : SPAREN SAVE ;    : UNSPAREN UNSAVE ;
 4 : UNBEDINGT ABS ;     : UND AND ;
 5 : HOCHST MAX ;     : MINDEST MIN ;     : REST MOD ;
 6 : ODER OR ;    : /REST /MOD ;    : 0SETZEN 0SET ;
 7 : 1SETZEN 1SET ;    : HIER HERE ;    : VERGESSEN FORGET ;
 8 : SCHLUSSEL CODE ;    : BESTANDIG CONSTANT ;
 9 : GANZE INTEGER ; : ORDNUNG ARRAY ; : IORDNUNG IARRAY ;
10 : SETZEN SET ;
11 : AUFS-LAUFENDE UPDATE ; : AUSRADIEREN ERASE-CORE ;
12 : LADEN LOAD ;    : ZURUCK CR ;
13 : SCHREIBEN TYPE ;     : BASIS BASE ;
14
15
16 ( THE SAME: DUP, MINUS, =L, BLOCK, ;S, F)    ;S
```

The control operation definitions (OB, DANN, SONST, BEGINNEN, ENDEN, TUN, SCHLINGE, +SCHLINGE) are not shown here; they are all short (one line), and exact copies of the English operations. (Incidently this particular vocabulary is a rough draft; we have not seen the results of the International Forth Standards Team, which is currently at work.) The word GERMAN is defined on the load screen, so that the Forth user can call in the German vocabulary when desired. The following session shows the entry and execution of a fibonacci

sequence program in German (until 16-bit overflow).

```
FORTH LOAD
OK
GERMAN
OK
: PRUFUNG 0 1 30 0 TUN VERT DUP . UBER + SCHLINGE ABWERFEN ABWERFEN ;
OK
PRUFUNG
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 1771
 28657 -19168 9489 -9679 -190 -9869 -10059
```

On a (slightly) more speculative note, why not extend this scheme to a computer assisted international language for computer conferences, electronic mail, and international data-base utilities? Clearly natural language is too free, and computer languages like BNF are too restrictive, to be feasible. But a hybrid, a vocabulary of several hundred unambiguous words (each used in one sense only), and perhaps some computer-oriented syntactical markers, should be enough for useful dialog within a particular interest area. If it works for two languages it should work as well for any number. The final test - whether international teams could collaborate, after minimal training - would take a few weeks programming at most, after the vocabularies and terminal interfaces had been determined.

FORTH MAILBAG

By Dave Bengel

The Forth Interest Group developed from several people in the San Francisco- San Jose area who have been working on Forth for the last year and a half. Until about six months ago most of these users were unaware of each other. Until the publication of the article by John James in Dr. Dobb's Journal (May 1978), about 80 percent of the group was from the Homebrew Computer Club - whose publication should also be watched for news concerning the F.I.G.

We now have nearly 200 names on the mailing list, and are receiving about six letters a day. The writers' main question is how to get a version for their machine.

We don't yet have a detailed description of the versions of Forth now available, nor is there a standard form of the language available for various CPUs. Intense work on implementations is now underway; e.g. the Forth Interest Group implementation workshop. We will keep you informed as more documentation and systems become available.

Your answers to the questionaire in this newsletter will help us keep a mailing list for interest-specific applications, documentation, CPU versions, etc. We appreciate any information you can send us, particularly about Forth versions or variants which are running or being developed, or any software we can publish. We need your contributions to this newsletter (which we hope will grow into a journal).

FIG LEAVES

IMPLEMENTATION WORKSHOP

1771

The Forth Interest Group will hold an implementation workshop, starting sometime in July. The purpose is to create a uniform set of implementations for common micros. The assembly listings which result will be available through F.I.G. to those who wish to distribute specific versions.

This will be a small group, no more than ten, with only one person for each machine. There will be approximately four all-day sessions, over six weeks. Implementers must have access to their target machine, with an assembler and editor; floppy or tape is not required. The meetings will be to share notes and specific guidance on implementation details. At the end of the workshop we should have uniform Forth versions for all the machines.

We now have implementers scheduled for 8080, 6502, PACE, and LSI-11. We need implementers for 6800, Z80, 1802, F8, System 3 (32), 5100, etc. We also need a project librarian with Forth experience.

If you are interested write to FORTH IMPLEMENTATION PROJECT, 20956 Corsair Blvd., Hayward, Ca. 94545.

CONTRIBUTED MATERIAL

Forth Interest Group needs the following material:

(1) Manuals available for distribution. We can purchase copies and distribute, or print from your authorized original.

(2) Name and address of Forth implementations for inclusion in our publications. Include computer requirements, documentation, and cost.

(3) Technical material for the forthcoming journal. Both expositions on internal features of Forth and application programs are needed.

(4) Users who may be referenced for local demonstration to newcomers, on a regional basis. Indicate interest area (i.e. personal computing, educational, scientific, industrial, etc.).

(5) Letters for publication in this newsletter.

(PAGE 8 AND 9 BLANK)

FORTH DIMENSIONS

AUGUST/SEPTEMBER 1978 VOLUME 1 NO. 2

CONTENTS

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

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 though 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

FOR NEWCOMERS

FORTH listings consist of sequences of "words" that execute and/or compile. When you have studied a glossary and a few sample listings, you should develop the ability to understand the action of new words in terms of their definition components. For the time being, we present a simplified glossary of the undefined words in this issue of FORTH DIMENSTIONS. For a fuller listing send for the F.I.G Glossary.

: xxx ..... ;
':' creates a new word named 'xxx' and compiles the following words (represented at ....) until reaching ';'. When 'xxx' is later used, it executes the words right after its name until the ';'.

CONSTANT VARIABLE
Each creates a new word with the following name, which takes its value from the number just before.

IF ELSE THEN
A test is made at 'IF'. If true, the words execute until the ' ELSE' and skip until THEN. If false, skip until ELSE and execute until THEN.

BEGIN END
At END a test is made; if false, execution returns to BEGIN; otherwise continue ahead.

DO LOOP LEAVE
At DO a limit and first index are saved. At LOOP, the index is incremented; until the limit is reached, execution returns to DO. LEAVE forces execution to exit at LOOP.

DUP DROP OVER SWAP ROT + - * /
These words operate on numbers in a stack just as then do in a HP calculator. If you like HP, you'll love FORTH.

>R R>
>R moves the top stack number to another stack. R> retreives it back to the original stack.

@ C@
@ Fetches the 16 bit contents of an address. C@ does the same for a byte. C@ may be also called B@ or \@.

! C!
These words store the second stack number at the memory address on the top of the stack. C! stores only a byte; it may be named B! or \! on some systems.

TYPE types a string by memory address and character count.

SPACE types a space.

CR types a carriage return/line feed.

MOVE moves within memory by addresses and byte count.

. prints a number.

.R prints a number in a tabulated column.

2+ adds two to the stack top number.

STATE is a variable, true when compiling.

(skips over comments until finding a ')'.

EXECUTE executes the word whose address is on the stack.

' finds the address of the next input name.

AND is a bitwise logical and.

, places a number in memory as part of compiling.

= > < are logical comparisons of stack numbers.

20 WORD fetches the next input word string.

HERE is a temporary memory workspace.

IMMEDIATE <BUILDS DOES> are too involved to discuss here. They are described in some detail in the text.

EDITORIAL

FORTH DIMENSIONS is dedicated to the promotion of extensible, threaded languages, primarily FORTH. Currently we are seeing a proliferation of similar languages. We will review all such implementations, referring to sources and availability.

Our policy is to use the developing "FORTH 77" International Standard as our benchmark.

Variant languages, such as STOIC, URTH, and CONVERS, will be evaluated on their advantages and disadvantages relative to FORTH. However, in evaluating languages named FORTH, we will note their accuracy in implementing all FORTH features. We expect complete versions named FORTH to contain:

1. indirect threaded code
2. an inner and outer interpreter
3. standard names for the 40 major primitives
4. words such as ;CODE, BLOCK, DOES>, (or ;:), which allow increased performance.

We hope to enable prospective users/purchasers to correctly select the version and performance level they wish, to foster long-range growth in the application of FORTH.

W.F.R.

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.

EXTENSIBILITY WITH FORTH

The purpose of any computer language (and its compiler or interpreter) is to bridge the gap between the "language" the machine understands (low level) and a language people understand (high level programming language). There are many choices for human-understandable languages: natural languages and artificial languages. The choice of language should allow convienient, terse, and unambiguous specification of the problem to be solved by the computer. Ordinarily only a few computer languages are available (e.g. BASIC, FORTRAN, APL). These were designed for certain classes of problems (such as mathematical equations) but are not suitable for others. The level of a language is a measure of suitability of that language for a particular application. The higher the level, the terser the program. By definition, [1] the highest level would allow a given problem to be solved with one operator (or command) and as many operands as there are input data required.

A natural language (e.g. English) might appear to be the best choice for a human-understandable computer language, and for some applications it may be. But natural languages suffer from three limitations: verbosity, ambiguity, and difficulty to decipher. This is partly because the meaning of a given word is dependent on its useage in one or more sentences (called "context sensitive") and because they require complex and nonuniform grammar roles with many exceptions. Specialized vocabularies and grammars permit terse and precise expression of concepts for restricted sets of problems. For example, [2] consider the following definition of a syllogism from propositional calculus:

$$((P1 \supset P2) \supset ((P2 \supset P3) \supset (P1 \supset P3)))$$

This sentence may be translated into English as "Given three statements which are true or false, if the truth of the first implies the truth of the second, this implies that if the truth of the second implies the truth of the third, then the truth of the first implies the truth of the third." Ambiguity is hard to avoid in most natural languages. The English phrase "pretty little girls school" (when unpunctuated) has 17 possible interpretations! (Try it.) [3]

As for the suitability of traditional programming languages (e.g. BASIC, FORTRAN, COBOL, PASCAL, APL) for "almost all technical problems", try coding the following "sentences" in your favorite computer language:

Quantum Mechanics:

$$H\psi = E\psi$$
where $H = -(\hbar^2/2m)\nabla^2 + V$
and E is the energy of the system

Electricity and Magnetism:

$$\nabla \cdot D = \rho$$
$$\nabla \cdot B = 0$$
$$\nabla \times E = -\frac{\partial B}{\partial t}$$
$$\nabla \times H = i + \frac{\partial D}{\partial t}$$

Matrix Algebra:

The trace of a matrix is equal to the sum of its eigenvalues.

Organic Chemistry:

$$3CH_3CH_2OH + Cr_2O_7^{--} + 8H^+ \rightarrow 3CH_3CHO(g) + 2Cr^{+++} + 7H_2O$$

Knitting:

K2 tog.3(5) times, *k1, p2, k1, k3, tog., k1,
p2, k1, p1, k1, p3 tog., k1, p1, p1* ; repeat
between *'s once more, k1, p2, k1, k3 tog.,
k1, p2, k1; k2 tog. 3(5) times; 47(51) sts.

Poetry:

Shakespearean sonnets are in iambic pentameter and
consist of three quatrains followed by a couplet.

FORTH is capable of being matched to each of the above relations at a high level. Furthermore, using the FORTH concept of vocabularies, several different applications can be resident simultaneously but the scope of reference of component words can be restricted (i.e., not global). This versatility is because the FORTH language is extensible. In fact the normal act of programming in FORTH (i.e., defining new words in terms of existing words) extends the language! For each problem programmed in FORTH, the language is extended as required by the special needs of that problem. The final word defined which solves the whole problem is both an operator within the FORTH language (which is also a "command") and the highest level operator for that problem. Further, the lower level words defined for this problem will frequently be useable for the programming of related problems.

It is true that popular computer languages allow new functions to be added using SUBROUTINES and FORTRAN-like FUNCTIONS. However these cannot be used syntactically the same as the operators in the language.

For example, let's assume a BASIC interpreter does not have the logical AND operator. To be consistent with similar, existing operators one would like to use "AND" in the following syntax:

A AND B

(A,B are any valid operands). Furthermore, one would want this new operator to have a priority higher than "OR" and lower than "NOT" so that

NOT A AND B OR C

would mean

(NOT A) AND (B OR C)

The only way to do this is to modify one's BASIC interpreter. Although not impossible, this is usually very difficult because of the following reasons:

(1) Most BASIC's are distributed without the interpreter source code (or not in machine-readable form). (2) One would have to learn this program and design a change. One modification might be "patched in", but many probably could not. Such a modified interpreter might not be compatible with future releases from the manufacturer. Errors might be introduced into other parts of the interpreter by this modification. (3) The process of changing is time consuming. Before one could try the new version, one would have to assemble, link, load, and possibly write a PROM. (How long would this take on your system?)

Another alternative would be to use a BASIC FUNCTION to add the AND operator, but it would have to be referenced as:

AND (A,B)

If NOT and OR were also FUNCTIONS, (NOT A) AND (B OR C) would have to be written as:

AND (NOT(A), OR(B,C))

This later form is lower level, less readable, and inconsistent with the intrinsic operators.

The addition of an AND operator in FORTH is as simple as any other programming addition; it would require one line of source code. The FORTH assembler could be used to take advantage of a particular processor's instruction, or the compiler could be used, resulting in machine transportability! Other differences from the example of modifying BASIC are: (1) Nothing existing in the FORTH system needs to be changed, so no learning is required, no errors are introduced to the existing system, and compatibility with future releases is preserved. The source of the FORTH kernel is not even necessary. (2) The new operator can be tried immediately after it is defined; if it is wrong it can be fixed before any further use is made of it. (3) This new operator is used exactly the same as any FORTH operator. So it may be mixed with existing operators in a totally consistant manner.

APL fans will point out that all the above is true for APL also. For something as simple as "AND" there is little difference. However, APL allows only monadic (single operand) and diadic (two operand) operators; FORTH operators can be written to accept as many operands as the programmer desires.

The previous discussion addressed the extension of the FORTH language, but it is almost as easy to extend the FORTH compiler! New compiler control structures (e.g., the CASE construct) can be added without changing any of the existing compiler. Or the existing compiler can be modified to do something different. To maintain compatibility with the existing compiler, the modifications could be part of a user- defined "vocabulary" so that both versions would be selectively available. Furthermore, one can write an entirely new compiler which accepts either the FORTH language or another language. (Complete BASIC interpreters have been written in FORTH.)

The choice of a computer language for a given application (including system development) should optimize the following attributes: (1) Be terse (i.e., the highest level for the application) (2) Be unambiguous (3) Be extensible (e.g., language, data types, compiler) (4) Be efficient (5) Be understandable (e.g., self documentation) (6) Be correct (e.g.,testing, proving assertions, consistency checks) (7) Be structured (e.g., structured programming, reentrant, recursive) (8) Be maintainable (e.g., modular, no side effects)

FORTH is a compromise among these goals, but comes closer than most existing programming languages.

;S KIM HARRIS

REFERENCES

1 Halstead, Maurice, Language Level: A Missing Concept in Information Theory, Performance Evaluation Review, ACM SIGMETRICS, Vol. 2 March '73.

2 McKeeman, Horning, and Wortman, A Compiler Generator, Prentice-Hall, 1970.

3 Brown, James Cooke, Loglan 1: A Logical Language, Loglan Institude, 2261 Soledad Rancho Road, San Diego, CA (714) 270-9773

GERMAN REVISITED

In the last issue of FORTH DIMENSIONS we showed how to create a bi-lingual (or multi-lingual) version of FORTH, and listed a simple program (set of FORTH definitions) for doing so. In respect to translation, there are three different classes of FORTH words:

(A) Those such as mathematical symbols which don't need to be translated.

(B) Words such as DO and IF which cannot be translated by a simple colon definition; the existing definitions must be re-copied and given German names. (all the definitions are short - one line - however)

(C) Other words, which could either be re-copied, or re-defined by a colon definition.

In any case, separate vocabularies can be used to prevent spelling clashes, no matter how many languages are spoken by one FORTH system. It can be possible to change languages as much as desired, even in the middle of a line.

The article stated that there was no run-time overhead. Such performance is possible, but the example given does have a run-time overhead of one extra level of nesting for each use of a word translated by a colon definition.

The following article by Bill Ragsdale is a more advanced treatment of language translation methods. It is written at the level of the FORTH systems programmer, and it uses a more standard FORTH version than the DECUS-supplied version which was used in the article which appeared in FORTH DIMENSIONS 1.

JOHN S. JAMES

FORTH LEARNS GERMAN

In the last issue of FORTH DIMENSIONS, we featured an article on natural language name *conversions for* FORTH. This article will add some additional ideas on the same topic.

First, the method shown (vectoring thru code) does have some run-time overhead. Also, some code definitions cannot execute properly when vectored in this manner, for example:

```
: R>   R>  ;
```

will pull the call of R> from the return stack and crash. We would ultimately like to translate names with:

1. Precisely correct operation during execution and compiling.

2. A minimum of memory cost.

3. A minimum of run-time cost.

4. A minimum of compile-time cost.

Let us now look at three specific examples to further clarify some of the trade-offs involved.

EXAMPLE 1 - COMPILING WORD

Let us see how UBER can be created to self-compile.

```
HEX
: D-E >R 2+ @ (2+ optional on some systems)
 STATE @ IF  (compiling) DUP 8 - C@ 80 <
  IF 2 - , ELSE (immediate ) EXECUTE THEN
 ELSE EXECUTE
  THEN ;
 : DO ENGLISH EMPLACE D-E ' , IMMEDIATE ;
   IMMEDIATE
 : UBER DO.ENGLISH OVER ;
 : LADEN DO.ENGLISH LOAD ; etc.
```

When building the translation vocabulary, the colon ':' creates the word UBER and then executes the immediate word DO.ENGLISH. DO.ENGLISH first emplaces the run-time procedure 'D-E' and then uses " ' , " to emplace the parameter field address of the next source word (OVER). Finally, the new word (UBER) is marked immediate, so that it will execute whenever later encountered.

Now we see how UBER executes. When it is interpreted from the terminal keyboard, 'D-E' will execute to fetch the emplaced PFA within the definitions of UBER (by R> 2+ @). After checking STATE the ELSE part will execute OVER from its parameter field address.

When UBER is encountered by the compiler in a colon definition, it will execute, as do all compiling words. Again R> 2+ @ will fetch the PFA of OVER to the stack. The check of STATE will be true and DUP 8 - C@ will fetch the byte containing the precedence bit. When compared to hex 80, a true will result for non-immediate, and 2 - , will compile the code field address.

However, if the word had been immediate (OVER isn't) the ELSE part will execute the word as in any compiling word.

The space cost of example 2 is 14 bytes per word (8 bytes per header and 6 bytes in the parameter field). The compile time cost is the execution of 'D-E.' There is no ultimate run-time cost in compiled definitions.

EXAMPLE TWO - <BUILDS - DOES>

Another way is to define a 'BUILDS-DOES' word E>G (English to German). It is then used to build a set of translation words similar to a FORTH mnemonic assembler.

```
: E>G   <BUILDS  ' , IMMEDIATE
        DOES>  @  STATE  @
        IF (compiling )  DUP 8 - C@ 80 <
            IF 2 - , ELSE EXECUTE THEN
          ELSE  EXECUTE  THEN  ;

E>G  UBER  OVER
E>G  LADEN  LOAD
E>G  BASIS  BASE     ..... etc.
```

E>G is a defining word that builds each German word (UBER) and emplaces the parameter field address of the English word (OVER) into the new parameter field (of UBER), and finally makes UBER immediate. When UBER is encountered by the outer interpreter, it does the DOES> part. The parameter of UBER, (the PFA of OVER) will be fetched and STATE tested. Since executing, the ELSE part will execute OVER from its parameter field address (as in the example 1).

When compiling, the DOES> part will be executed, again similarly to 'D-E' as in Example 1. The space cost of the BUILDS-DOES method is 10 bytes per word (8 in the header, 2 in the parameter field). The compile time is the same as in Example 1 and there is no run-time cost.

EXAMPLE THREE - RENAME

This last method is the most fool-proof of all. We will just re-label the name field of each resident word to the German equivalent.

```
: RENAME  ' 8 - DUP C@ 60
   AND (precedence bit  )
   20 WORD HERE C@ +
   HERE C! (store into length)
   HERE SWAP 4 MOVE ;

     ( overlay old name )

   RENAME OVER UBER

   RENAME LOAD LADED

   RENAME BASE BASIS
```

This method extracts the precedence bit of the old (English) definition and adds it to the length count of the new (German) name. The new name is then overwritten to the old name field. There is no space or time cost!! The dictionary is now truely translated.

A final caution is in order for Examples 1 and 2. Some FORTH methods may still give trouble. If you should try:

```
UBER
```

you will find the PFA of UBER which is a translating definition, and not the ultimate run-time procedure, (which is really in OVER). This would have disasterous results if you were attempting to alter what you thought was the executing procedure, and you were really altering the compiling word. For this reason, the method of Example 3 is the only truly 'fool-proof' method. The renaming method has the added use of allowing you to change names in your running system. For example, it is likely that the old <R will be renamed >R in the international standard FORTH-77. You can simply update your system by the use of the word RENAME.

;S W.F. RAGSDALE 8/27/78

THREADED CODE

Bell (1) and Dewar (2) have described the concepts of Threaded Code (also called Direct Threaded Code, or DTC), and an improvement called Indirect Threaded Code, or ITC. DTC was used to implement Fortran IV for the PDP-11, and ITC was used for a machine- independent version of Spitbol (a fast form of the string-processing language Snobol). Forth is a form of ITC, but different from the scheme presented in (2).

In DTC, a program consists of a list of addresses of routines. DTC is fast; in fact, only a single PDP-11 instruction execution is required to link from one routine to the next (the instruction is 'JMP @(R)+', where 'R' is one of the general registers). Overall, DTC was found to be about three percent slower than straight code using frequent subroutine jumps and returns, and to require 10-20 percent less memory. But one problem is that for each variable the compiler had to generate two short routines to push and pop that variable on the internal run-time stack.

In ITC, a program is a list of addresses of addresses of routines to be executed. As used in (2), each variable had pointers to push and pop routines, followed by its value. The major advantage over DTC is that the compiler does not have to generate separate push and pop routines for each variable; instead these were standard library routines. The compiler did not generate any routines, only addresses, so it was more machine independent. In practice, ITC was found to run faster than DTC despite the extra level of indirection. It also used less memory.

Forth is a form of ITC, with additional features.

Forth operations are lists of addresses pointing into dictionary entries. Each dictionary entry contains:

(A) Ascii operation name, length of the name, and precedence bit; these are used only at compile time and will not be discussed further.

(B) A link pointer to the previous dictionary entry. (This is used only at compile time.)

(C) A pointer called the code address, which always points to executable machine code.

(D) A parameter field, which can contain machine instructions, or Forth address lists, or variable values or pointers or other information depending on the variable type.

By the way, virtually everything in Forth is part of a dictionary entry: the compiler, the run-time routines, the operating system, and your programs. In most versions, only a few bytes of code are outside of the dictionary.

The code address is crucial; this is the 'indirect' part of ITC. Every dictionary entry contains exactly one code address. If the dictionary entry is for a "primitive" (one of the 40 or so operations defined in machine language), the code address points two bytes beyond itself, to the parameter field, which contains the machine-language routine.

If the dictionary entry is for a Forth higher-level operation (a colon definition), the code address points to a special "code routine" for colon definitions. This short routine (e.g. 3 PDP-11 instructions) nests one level of Forth execution, pushing the current 'I' register (the Forth "instruction counter") onto a return-address stack, then beginning Forth execution of the address-list in the new operation's parameter field.

If the dictionary entry is for a variable, then the code address points to a code routine unique to that variable's type. The parameter field of a variable may contain the variable's value - or pointers if re-entrant, pure-code Forth is desired.

Results of Forth-type ITC include:

(A) Execution is fast, e.g. two PDP-11 instruction executions to transfer between primitives, about ten to nest and un-nest a higher-level definition. (Because of the pyramidal tree-structure of execution, the higher-level nesting is done less often.) Yet the language is fully interactive.

(B) Forth operation names (addresses) are used exactly the same regardless of whether they represent primitives or higher-level definitions (nested to any depth). Not even the compiler knows the difference. In case run-speed optimization is desired, critical higher-level operations (such as inner loops) can be re-coded as primitives, running at full machine speed, and nothing else need by changed.

(C) Forth code is very compact. The language implements an entire operating system which can run stand-alone, including the Forth compiler, optional assembler, editor, and run-time system, in about 6k bytes. (Forth can also run as a task under a conventional operating system, which sees

it as an ordinary assembly-language program, and Forth can link to other languages this way.) Code is so compact that application-oriented utility routines can be left in the system permanently, where they are immediately available either as keyboard commands or instructions in programs, and they are used in exactly the same way in either case. No linkage editing is needed, and overlays are unusual.

REFERENCES

(1) Bell, James R. Threaded code. C. ACM 16, 6 (June 1973), 370-372.
(2) Dewar, Robert B. K. Indirect threaded code. C. ACM 18, 6 (June 1975), 330-331.

STAFF

The volunteer staffing of FORTH DIMENSIONS is a bit fluid. For this issue, our staff consisted of :

EDITOR	JOHN JAMES
CONTRIBUTORS	KIM HARRIS W.F.RAGSDALE
TYPESETTING	TOM OLSEN JOHN JAMES
ARTWORK	ANNE RAGSDALE
CIRCULATION	DAVE BENGEL
DATA PROCESSING	P D P -11

FORTH DEFINITION

FORTH is the combination of an extensible programming language and interactive operating system. It forms a consistent and complete programming environment which is then extended for each application situation.

FORTH is structured to be interpreted from indirect, threaded code. This code consists of sequences of machine independent compiled parameters, each headed by a pointer to executable machine code. The user creates his own application procedures (called 'words'), from any of the existing words and/or machine assembly language. New classes of data structures or procedures may be created; these have associated interpretive aids defined in either machine code or high level form.

The user has access to a computation stack with reverse Polish conventions. Another stack is available, usually for execution control. In an interactive environment, each word contains a symbolic identifier aiding text interpretation. The user may execute or compile source text from the terminal keyboard or mass storage device. Resident words are provided for editing and accessing the data stored on mass storage devices (disk, tape).

In applications that are to run 'stand-alone', a compact cross-compiled form is used. It consists of compiled words, interpretive aids, and machine code procedures. It is non-extensible, as the symbolic identifiers are deleted from each word, and little of the usual operating system need be included.

;S W.F.R. 8/26/78

NOTES

- The second meeting of the FORTH International Standards Team will occur in Los Angeles on October 16-19. Contact FORTH Inc. for additional information.

- A partially micro-coded FORTH-like language is described in "Threaded Code for Laboratory Computers" by J.B. Phillips, M.F. Burke, and G.S. Wilson, Dept. of Chemistry, University of Arizona, Tucson, AZ 85721. The article is published in Software - Practice and Experience, Volume 8, pages 257-263. Implementation is on a HP2100. The article also describes the advantages of threaded languages for laboratory applications.

- A "form" of FORTH for the Apple and PET 6502 based computers is available from Programma Consultants, 3400 Wilshire Blvd., Los Angeles, CA 90010. We have not used these enough to review them for this issue but they have been shipped and do work. For more information write to Programma Consultants or watch future issues of FORTH DIMENSIONS.

- FORTH Inc. is looking for a programmer with some systems-level experience using FORTH or similar languages. Interested persons should contact FORTH Inc., 815 Manhattan Avenue, Manhattan Beach, California 90266, (213) 372-8493 .

HELP

SCR # 6

```
 0     THE 'HELP' COMMAND IS PROBABLY THE MOST USEFUL OPTION FOR
 1 A FORTH SYSTEM.  IT ALLOWS YOU TO VIEW THE DICTIONARY WORDS
 2 AND LOCATE THEM IN MEMORY.  WHEN YOU ARE TESTING NEW
 3 DEFINITIONS, IT WILL SHOW RE-DEFINITIONS.  IT IS A WAY TO
 4 LOCATE WHERE A MISSING WORD SHOULD BE, BUT ISN'T.
 5
 6     IF YOU MAKE A COMPILE ERROR FROM DISC, 'HELP' WILL SHOW
 7 THE WORD IN WHICH THE ERROR OCCURED.
 8
 9     YOU SHOULD MODIFY THE FOLLOWING DEFINITIONS TO THE FORMAT
10 YOU WANT.  FOR OBJECT CODE EXAMINATION, I LIKE THE CODE FIELD
11 ADDRESSES AS SHOWN, SINCE THIS IS WHAT RESULTS IN THE COMPILED
12 CODE.   FOR A QUICK SNAP-SHOT OF THE DICTIONARY, I JUST PRINT
13 THE LENGTH AND NAMES.
14
15     JUST TYPE 'HELP' AND HIT THE 'BREAK' KEY TO STOP.
```

SCR # 7

```
 0 (  HELP  )                HEX
 1 00      CONSTANT  LAST.LINK    ( IS $8000 ON MICRO-FORTH )
 2  4      CONSTANT  #/LINE       ( WORDS PRINTED PER LINE  )
 3
 4 : .NAME              ( ENTER WITH ADDRESS OF LENGTH BYTE )
 5    DUP  C@  7F  AND  DECIMAL  3  .R  SPACE  1+  3  TYPE SPACE  ;
 6
 7 : .CODE-ADDRESS      ( ENTER WITH ADDRESS OF LENGTH BYTE )
 8     6  +  HEX  5  .R  SPACE  ;
 9
10 : .HEADER            ( ENTER WITH ADDRESS OF LENGTH BYTE )
11     DUP  .NAME  .CODE-ADDRESS  ;
12
13 : ?TERMINAL  0  ;  ( USER'S MACHINE DEPENDENT TERMINAL BREAK )
14              ( RETURN '00' FOR NO BREAK, AND '01' FOR A BREAK )
15 8 LOAD      ;S    8/27/78    WFR
```

SCR # 8

```
 0 (  HELP, CONT.  )
 1
 2 : .LINE           ( PRINT A LINE OF NAMES AND CODE ADDRESSES )
 3    #/LINE  0      ( ENTER WITH ADDRESS OF LENGTH BYTE        )
 4    DO  DUP  .HEADER  SPACE    4  +  @  DUP  LAST.LINK  =
 5        IF  LEAVE  THEN  LOOP  ;     ( EXIT WITH NEXT ADDRESS )
 6
 7 : HELP          ( PRINT DICTIONARY FROM TOP CURRENT WORD DOWN )
 8           ( TO BOTTOM.  FORMAT IS LENGTH COUNT, 3 LETTERS OF )
 9           ( NAME, AND CODE FIELD ADDRESS.  WILL TERMINATE    )
10           ( UPON LAST LINK VALUE OR A TERMINAL BREAK.        )
11    BASE  C@  >R    CURRENT  @  @
12    BEGIN  CR  .LINE  DUP  LAST.LINK  =  ?TERMINAL  +
13       END   DROP ( LAST LINK )  R>  BASE  C!  ;
14
15 DECIMAL     ;S     8/28/78     WFR
```

```
HELP
  4 HEL  1EBB    5 .LI  1E90    7 .HE  1E80   13 .CO  1E68
  5 .NA  1E43    6 0/L  1E39    9 LAS  1E2F    4 TAS  1E25
  4 BOO  1B8B    2 ->   1B6A    4 TUB  1B57    3 TTY  1B44  OK
```

ABOVE WE SEE AN EXAMPLE OF THE LOADING OF THE 'HELP' COMMAND FROM DISC. IT THEN IS TESTED, AND DUMPS THE DICTIONARY. WE SEE THE LISTING OF 'HELP' AND THE WORDS IS USES LISTING CONTINUES INTO THE RESIDENT DICTIONARY.

GOOD LUCK, WFR

MANUALS

DECUS PDP-11 FORTH
by Owens Valley Radio Observatory, California Institute of Technology, Martin S. Ewing. (alias The Caltech FORTH Manual) Available from DECUS, 129 Parker Street, PK3/E55, Maynard, Mass. 01754. Ordering information: Program No. 11-232, Write-up, $5.00 .

FORTH Systems Reference Manual
W. Richard Stevens, Sep 76. Kitt Peak National Observatory, Tucson, AZ 85726. (NOT FOR SALE)

LABFORTH
An Interactive Language for Laboratory Computing, Introductory Principles, Laboratory Software Systems, Inc., 3634 Mandeville Canyon, Los Angeles, CA 90049. $8.00 .

STOIC
(Stack Oriented Interactive Compiler) by MIT and Harvard Biomedical Engineering Center. Documentation and listings for 8080 from CP/M Users Group, 164 west 83rd Street, New York, N.Y. 10024. $4.00 membership, $8.00 per 8" floppy, 2 floppies needed.

CONVERS
The Digital Group, Box 6528, Denver, CO 80226 Manual: DOC-CONVERS $12.50 .

URTH
(University of Rochester FORTH), Tutorial Manual, Hardwick Forsley, Laboratory for Laser Energetics, 250 E. River Rd., Rochester, NY 14621 .

microFORTH Primer
FORTH, Inc. 815 Manhattan Ave., Manhattan Beach, CA 90266 (moving soon) $15.00 .

(Page 21, 22 Blank)

FORTH DIMENSIONS

OCTOBER/NOVEMBER 1978 VOLUME 1, NO. 3

CONTENTS

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

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 though 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.

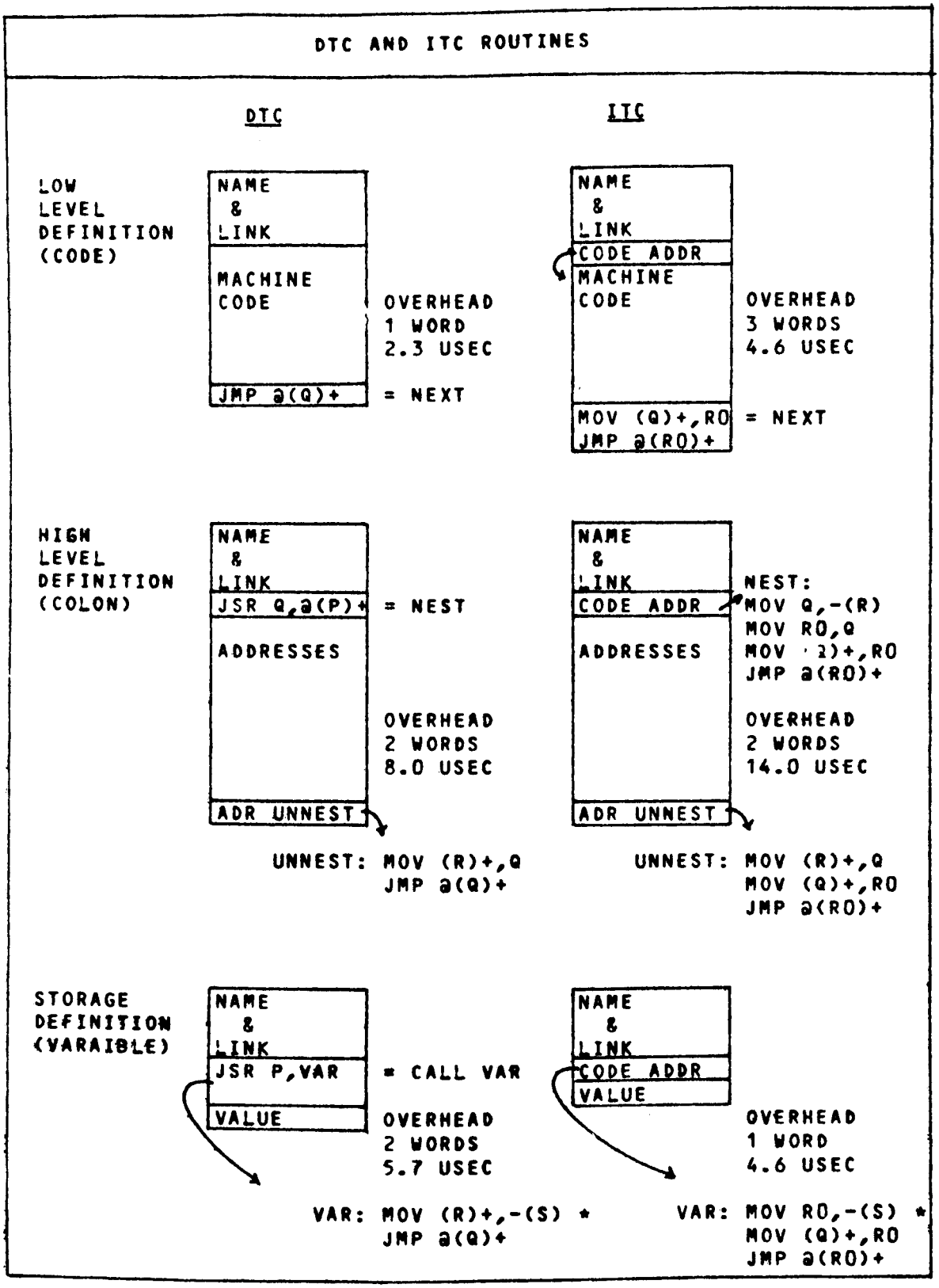
S is the stack pointer (R4).

R is the return stack pointer (R6).

P is the program counter (R7).

R0 is a temporary register assumed to be available.

-->



* 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

	DTC	ITC	DTC ADVANTAGE
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

	SIZE ADVANTAGE	SPEED ADVANTAGE
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	.7 words	2.4 usec

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 independance because no machine code appears in the code generated by the compiler. But even this advantage is illusionary 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
David J. Sirag
Laboratory Software Systems, Inc.

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

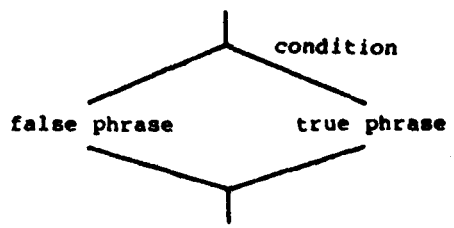
In FORTH, this structure takes the form:

```
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:



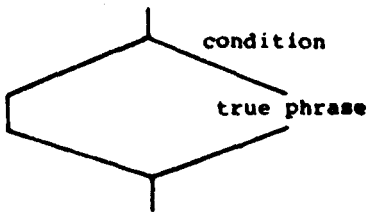
words following ENDIF (or THEN)

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

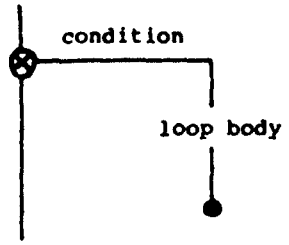
a vertical line is drawn as shown:



-->

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.

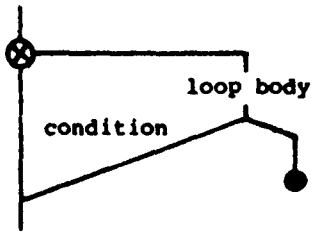
3. The "condition" is again evaluated. Its outcome might have changed during the execution of the loop statement.

   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   .
```

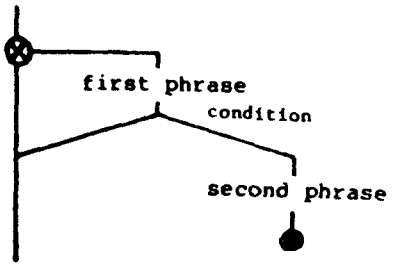
The loop body is always executed once, and is repeated as long as condition is false. The D-chart symbol for this structure would be:



A more general case is

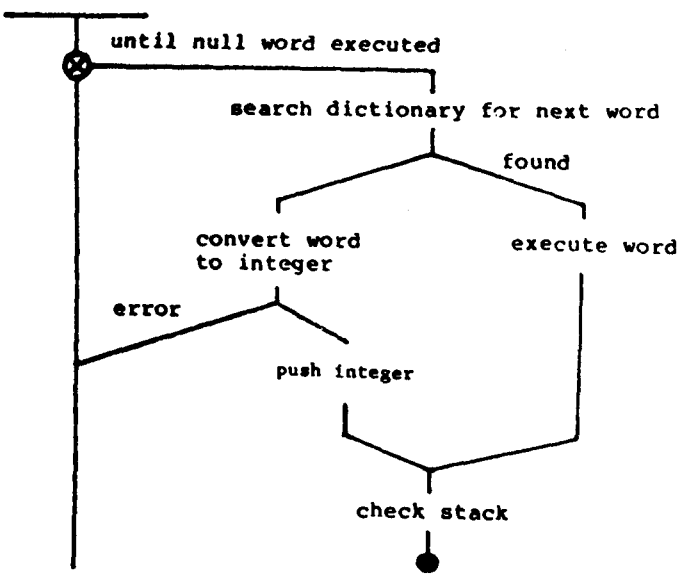
```
          BEGIN   first phrase
condition IF   second phrase
          AGAIN
```

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 F.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:

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 form 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 "PARN" 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

SYSTEM LANGUAGE 1

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 is 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 exeuction 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 mneumonic 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 Ø 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,Ø        ;stack in 16-bit
        JMP HPUSH      ;format
```

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

```
4ØØØ DB ØF ØF D2 ØØ 4Ø DB EC 6F
     26 ØØ C3 41 ØØ
```

This is exactly what the Assembly code would produce if ORG'ed at 4ØØØH 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 hign 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 Ø DO KEY 7F AND DUP ØD =
             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 Ø 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 Ø         ;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 ØD        ;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 OPCODE 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 RBM

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 disking 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 echo'ed to CRT to signal completion.

;s RBM Oct. 1978

FORTH DIMENSIONS

DECEMBER 1978/JANUARY 1979 VOLUME 1 No. 4
(Published July 1979)

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

PUBLIC MEETINGS OF F.I.G

The Forth Interest Group is pleased to announce a public meeting series. We will meet on the fourth (!) Saturday of the month in Hayward, Ca., in the Special Events Room of the Liberty House Department Store, in the Southland Shopping Center (800 Southland Mall)

This room is on the third floor rear. The formal meeting begins at 1:00 PM, but we gather for lunch about 12 Noon.

The specific dates are July 28, Aug 25, Sept 22, Oct 27, and Nov 24. A single technical topic will be presented in detail, with workshop sessions on the fig-FORTH model, and any topics of immediate interest. In July, Dave Lyons will present the ascii memory dump of Forth which is part of his 6800 version. This Forth program shows the entire language map on one sheet of paper!

PUBLISHERS COMMENTS

The issues of Forth Dimensions have been scheduled for two month intervals, but have been at intervals determined by the overall activities of the steering committee. In the past months we have participated in the International Standards Team, given conference papers the Fourth West Coast Computer Faire, attended the Forth Users Meeting in Utrecht, Holland.

These events have been outwardly reflected in the large gap since Issue 3. For the near term, hope is in sight!. We have Issue 5 ready for publication in three weeks, and Issue 6 should occur within a month. This will wrap up Volume 1. At this time we will evaluate our efforts, and plan for future activities. Member comment will be quite apropos during this period and will help shape future application of effort.

Thanks are due the membership for their patience during this period.

STAFF

The volunteer staffing of Forth Dimensions is a bit fluid. For this issue, our staff consisted of:

EDITOR	Bill Ragsdale
REVIEW	Dave Boulton
CONTRIBUTORS	Paul Bartholdi D. W. Borden
TYPESETTING	GLG Secretarial and Vydec
ARTWORK	Anne Ragsdale
CIRCULATION	6502 TIM and Persci

HISTORICAL PERSPECIVE

FORTH was created by Mr. Charles H. Moore in about 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of this 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/I 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 vendors.

The Forth Interest Group is centered in Northern California, although our membership of 450 is world-wide. It was formed in 1978 by local Forth programmers to encourage use of the language by the interchange of ideas through seminars and publications. All effort is on a volunteer basis and the group is affiliated with no vendors.

;S FIG

THE "TO" SOLUTION

Paul Bartholdi
Observatoire De Geneve
CH-1290-Sauverny
Switzerland

At the Catalina standardization meeting, Chuck Moore suggested rapidly the "TO" construct to aleviate some, if not all, of the difficulties associated with address manipulation. This new construction seems to me extremely powerful. It should considerably decrease the number of errors and improve the readability of programs. It is very easy to understand and to use (much easier in fact than the constructs it is replacing!). Its implementation is also trivial but the lack of experience in using it may hide some difficulties. These notes try to sketch its potentials.

1. The " TO " concept

The basic concept is the following: The code associated with VARIABLEs is divided into two exclusive parts.

- The first one (or " fetch code ") is identical with the one associated normally with CONSTANT . It pushes on the stack the value (byte, word or words) in the parameter field.
- The second one (or " store code ") is new. It transfers the value (byte, word or words) from the stack into the parameter field, it is equivalent to <variable-name> ! (or C! or D! or F!).

The choice between the two codes depends on a state variable (which will be called %VAR hereafter) that has two possible states: 0 (or fetch state) and 1 (or store state).

If %VAR = 0 the first code is executed and %VAR is unchanged.

If %VAR = 1 the second code is executed and % VAR is immediately returned to 0.

The operator " TO " sets %VAR to 1, forcing the next-called variable to store the content on top of the stack in its parameter field instead of fetching it.

2. Examples using " TO "

We suppose three single variables called A , B and C three double variables F , G and H , and three floating variables X , Y and Z . Then the following half-lines are equivalent. The first column assumes the old definitions of variables, the second one uses the new concept.

	OLD :	NEW :
i)	A @ B @ + C !	A B + TO C
ii)	F D@ G D@ D- H D!	F G D- TO H
iii)	X F@ Y F@ F* Z F!	X Y F* TO Z

3. VARIABLEs or CONSTANTs ?

Some advantages of the new concept are quite evident from the previous examples. Just before the Catalina meeting, I came to the conclusion that variables should be dropped altogether. If A , B , ... Z had been constants in the previous examples then, using P and ! , the three lines would have been coded

```
 i)  A  B  +  '  C  !
ii)  F  G  D- '  H  D!
iii) X  Y  F* '  Z  F!
```

which is already better than the "old" above.

The difficulty starts with ARRAYs. If they push values onto the stack instead of addresses, then it becomes much more difficult to store values at the right places.

Note that ' C (or H or Z) takes two words but is really, at execution time, a single operator (LIT <address-of-C>). " TO " is then the shortest solution in terms of memory space, and more or less equivalent to the CONSTANT solution in terms of the time used.

4. The " address " problem

But the main advantage of " TO " in this context are the following:

- In the general sense a " CONSTANT " should be used as such, and never (or hardly ever) be changed. In particular it may reside in PROM . I suggest then to keep the constants as such, its associated code ignoring the value of %VAR . We then should redefine the variables to check %VAR and behave accordingly.

- One of the main unresolved points at Catalina was the definition and use of addresses for variables in the general sense. One consensus was obtained in September at the Geneva meeting, that is, as far as possible, addresses should be omitted. The " TO " concept solves this requirement admirably. No addresses are

ever put on the stack, or manipulated explicitly. Then byte or word addressing is irrelevant. It is taken care of at the system level only, in the code of the second part (the "store code").

5. Portability

Because of this, FORTH programs become more portable. " TO " replaces all fetch and store operators which would or would not be distinct, which would work on byte or "position" or word addresses. Transporting a program from a micro to a large CDC implies now much less adaptations.

6. Clarity - security

Using less operators, in fact the strict minimum, is probably one of the best ways of improving clarity. Note also that " TO " appears as an infix operator to the programmer (and reader!). In terms of security, " TO " implies that only the parameter field of variables can be changed. Other addresses are not (at least directly) accessible.

This is certainly a tremendous gain for some environments.

7. The ARRAY problem

The use of " TO " with ARRAYs is not as simple as with variables, but still quite practicable. Note first that anything but a variable can stay between " TO " and the variable's name. If C is defined as an ARRAY (with the double associated code) then

```
A TO 4 C (instead of A @ 4 C ! )
DO I TO I C LOOP
(instead of DO I I C ! LOOP )
```

are quite alright and extend all the previously noted advantages of " TO ". But the programmer must take care that the index must not contain a variable. For example:

```
4 TO B ... A TO B C
```

will put the value of A into B instead of into C The necessary form should then be

```
A B TO C
```

which is surely less pleasant because of its asymmetry.

8. Fetching and storing inside a disk block

The problem extends of course to "virtual arrays" like the disk blocks. In this context, direct access should be considered separately from Data Management. The double code concept associated with " TO " should be extended to the Data Management operators. For the direct manipulation of data inside a disk block, I suggest the creation of a new operator, with the double code, associated with the form of

```
<relative-word-address>
<bloc-number> BLOC
```

(BLOC is of course just a provisional name!)

Example:

```
4 12 BLOC 5 + TO 5 12 BLOC
```

instead of

```
12 BLOCK DUP 4 + @ 5 +
SWAP 5 + !
```

or

```
12 BLOCK 4 + @ 5 + 12
BLOCK 5 + !
```

9. General access to the (whole) memory

No good FORTH programmer would ever accept to be strongly restricted in his access to the memory. But if the " TO " concept is really accepted, then all the @ and ! operators should disappear. I suggest, instead, to add (its usage could be restricted) a generalized array called MEMORY (or any equivalent) with once more the double code associated with it. Then <address> MEMORY would either fetch from or store into the real address.

As an example, we would have

```
Ø TO 15317 MEMORY instead of
Ø 15317 ! or

DO I MEMORY S. LOOP instead of
DO I @ S. LOOP etc.
```

MEMORY is clearly equivalent to @ and ! depending on $VAR.

10. Generalization of address matching: virtual arrays

The previous points 8 and 9 suggest a further generalization that may solve another problem we have had since the beginning of FORTH: the use of arrays as parameters for a procedure. It was generally solved by putting the address of the first element on the stack, and doing explicit address arithmetic in the procedure (see the FFT of Jim Brault for example). This is certainly neither clean nor fast.

What I propose is the following: A virtual array (VARRAY , DVARRAY , FVARRAY , CVARRAY etc.) behaves like an array, but does not reserve space, except for a pointer to the real array. The link between the virtual and any portion of the memory is established by the word MATCH .

For example:

```
1ØØ ARRAY CUSTOMER
1ØØ ARRAY STAR
    VARRAY NUMBER
  , CUSTOMER MATCH NUMBER
```

associates the real array CUSTOMER with the virtual array NUMBER .

Then <i> NUMBER will be equivalent to <i> CUSTOMER.

Remember that, as previously,

<i> NUMBER pushes the value of the i^{th} STAR or CUSTOMER on the stack.

and <m> TO <i> NUMBER will store <m> into the i^{th} STAR or CUSTOMER .

At any time, CUSTOMER can be replaced by STAR , (and vice versa) by

```
' STAR MATCH NUMBER
```

In this way, MEMORY is defined by

```
          VARRAY   MEMORY
       ø  MATCH    MEMORY
```

```
;S  PB
```

```
SCR # 150
  0 (  Example of the creation of TO  )
  1 HERE  0  ,  CONSTANT  ZVAR    1  ZVAR  SET  TO
  2 : VAR   CONSTANT  ;CODE  INB,  ZVAR  LDA,
  3        A0 IF,  B I) LDA,  PUSH,
  4             ELSE,  CLA,  ZVAR STA,  S) LDA,  B I) STA, POP,
  5             THEN,
  6 : DVAR   CONSTANT  ,  ;CODE  INB,  ZVAR LDA,
  7        A0 IF,  B LDA,  DSP,  A I) DLD,  S) STB,  PUSH,
  8             ELSE,  CLA,  ZVAR STA,  ..T STB,  S) DLD,
  9                    ..T I) DST,  POP.,
 10             THEN,
 11 : ARRAY   0  CONSTANT  DP  +!  ;CODE  INB,  S) ADB,  ZVAR LDA,
 12        A0 IF,  B I) LDA,  PUT,
 13             ELSE,  CLA,  ZVAR STA,
 14                    S1) LDA,  B I) STA,  POP.,
 15             THEN,

SCR # 151
  0 : 2ARRAY   0  CONSTANT  DUP  +  1+  DP  +!
  1       ;CODE  INB,  S) ADB,  S) ADB,  ZVAR LDA,
  2         A0 IF,  B I) DLD,  S) STB,  PUSH,
  3              ELSE,  CLA,  ZVAR STA,  ..T STB,
  4                     ISP,  S) DLD,  ..T I) DST,  POP.,
  5              THEN,
  6 : VARRAY  0 CONSTANT  ;CODE  INB,  B I) LDB,  S) ADB,  ZVAR LDA,
  7         A0 IF,  B I) LDA,  PUT,
  8              ELSE,  CLA,  ZVAR STA,  S1) LDA,  B I) STA,  POP,
  9              THEN,
 10 : DVARRAY  0 CONSTANT ;CODE INB, B I) LDB, S) ADB, S) ADB,
 11         ZVAR LDA,  A0 IF, B I) DLD,  S) STB,  PUSH,
 12              ELSE,  CLA,  ZVAR STA,  ..T STB,
 13                     ISP,  S) DLD,  ..T I) DST, POP.,
 14              THEN,
 15 FORTH   IMP  '  : MATCH  '  !  ;    IMP  '
```

Pauls' example of the use of TO will be presented in the next issue of Forth Dimensions. It utilizes Knuths' example for the calculation of the dates of Easter.

FORTH IMPLEMENTATION PROJECT

In June of 1978, the Forth Interest Group held its' first public meeting. With only mimimal publicity, we had in excess of 40 people attend. We had intended to offer educational assistance in using Forth. However, we found everyone was enthusiastic to learn Forth, but only five had access to running systems.

FIG then surveyed for vendor availability of the language. We found there were numerous mini-computer versions at educational and research institutions, all directly decended from Mr. Moore's word at NRAO. Of course, Forth, Inc offers numerous commercial systems.

None of these systems were available for personal computing. It appeared unlikely that this need would be met in the forseeable future. Our conclusion was that a suitable model should be created, and transported to individual micro-computers. Thus was born the Forth Implementation Team (FIT).

This team was proposed as a three tier structure. The first tier had several experienced Forth systems programmers, who would provide the model and guide the implementation effort. The next tier was the most critical. It was composed of systems level programmers, not necessarily having a background in Forth. They were to transport the common language model to their own computers by generating an assembly language listing that followed the model. Their results would be passed to the distributors that form the third tier. These distributors would customize for specific personal computer brands. Finally, the users could have access to both source and object code for maintenance.

Space doesn't permit inclusion of the FIT project, itself. The project was detailed as one of the six Forth conference papers at the Fourth West Coast Computer Faire, May 1979 in San Francisco.' [1] The result is that FIG now offers the Installation Manual with glossary and Forth model ($10.00) and assembly language listings for numerous computers

($10.00 @) Included are: 8080, PDP-11, PACE, 9900, 6800, and soon 6502, and Z-80.

Note that FIG offers these listings which still have to be edited into machine readable form, customized, and assembled for specific installations. We hope that local teams share the effort and then distribute for others.

Reports of installations are beginning to come in from the USA and Europe. We sincerely hope this work will give a benchmark of quality and uniformity that will raise the expectation of all users.

FIG would like to thank the following members of the Implementation Team who have devoted a major part of nine months' spare time to this effort.

Dave Boulton	Instructor
John Cassady	8080
Gary Feierbach	Comp. Aut.
Bernard Greening	Z-80
Kim Harris	Librarian
John James	PDP-11
Dave Kilbridge	PACE
Dave Lion	6800
Mike O'Malley	9900
Bill Ragsdale	Instructor
Bob Smith	6800
LaFarr Stuart	6800

[1] Ragsdale, William F.
"Forth Implementation, A Team Approach"
The Best of the Computer Faires, Vol IV
from: Computer Faire ($14.78, USA)
333 Swett Road, Woodside, CA 94062

FORTH INTERNATIONAL STANDARDS TEAM

For several years the Forth Users Group (Europe) has sponsored a team working toward a standards publication for Forth. The 1977 meeting (Utrecht) produced a working document FORTH-77. Attendees included European educational institutions and Forth, Inc.

In October, 1978, an expanded group met at Catalina Island (Calif.). Attendees included Forth Users Group (Neiuwenhuizen, Bartholdi), Forth, Inc. (Moore, Rather, Sanderson), FIG (James, Boulton, Ragsdale, Harris), Kitt Peak (Miedaner, Goad, Scott), U of Rochester (Forsley), SLAC (Stoddard), and Safeguard Ind. (Vurpillat).

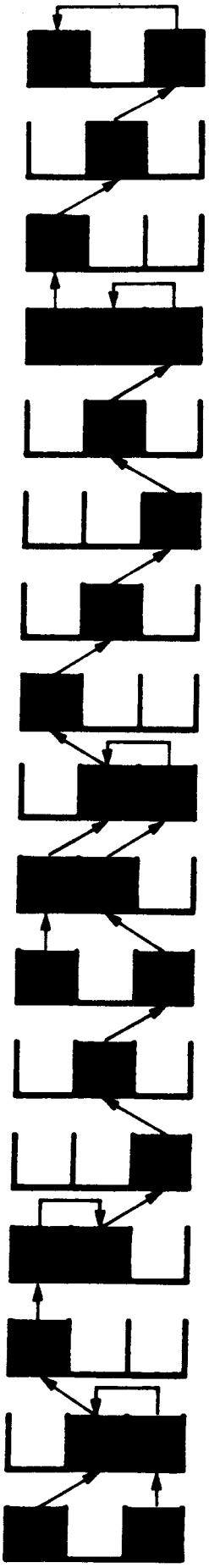
The document resulting from this four day meeting has been released as FORTH-78. This document is becoming a good reference guide in evaluating the consistency and completeness of particular Forth systems. It is available from FIST to participating sponsors. (See below.)

A major benefit of the team meeting was the development of close communications between major users. For example, FIG is learning from the multi-tasking of U of R, Kitt Peak and Utrecht are running fig-FORTH, and we have adopted the security package pioneered in Europe. None of these events would have been likely without the contacts begun at Catalina.

The Team has announced the next Standards Meeting for October 14 thru 18, 1979, again at Catalina. The team agreed on an orginizational budget of $1000.00, to be met by $30. contributions by sponsors (individuals and companies).

These funds will be used solely to defray organizing costs of the annual meeting and distribution of the working documents to participants. Those considering participating should become Team Sponsors by remiting as given below. Sponsors will receive the just released FORTH-78, and all Team mailings.

Please remit to FIST, %Carolyn Rosenberg, Forth, Inc. 815 Manhattan Ave., Manhattan Beach, CA 90266.



FORTH DIMENSIONS

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

VOLUME I

Numbers 1 - 6

I/1-6/500

FORTH DIMENSIONS INDEX
VOLUME I,II,III

INDIVIDUAL WORDS DEFINED

MISC. ENTRIES

SCREENS OF FORTH CODE

INDEX COMPILED COURTESY OF
M. TASSANO
936 DELAWARE WAY
LIVERMORE,CA. 94550

FORTH DIMENSIONS

JUNE/JULY 1978 VOLUME 1 NO. 1

EDITORIAL:
WHAT IS THE FORTH INTEREST GROUP?

The Forth Interest Group, which developed in the fertile ground of the computer clubs of the San Francisco Bay Area, grew in a few months from nothing to where we are now getting several letters a day from all over the country. With this increasing public interest we need to let people know what we are doing and why, what we would like to see happen, how others can be involved, and what we can and cannot do.

We are involved because we believe that this language can have a major effect on the usefulness of computers, especially small computers, and we want to see it put to the test. Increasingly software is becoming the critical, limiting factor in the computer industry. Large software projects are especially difficult to develop and modify. Few are happy with prevailing operating systems, which are huge, hard to understand, incompatible with each other, and without unity of design.

The Forth language is its own operating system and text editor. It is interactive, extensible (including user-defined data types), structured, and recursive. Code is so compact that the entire system (mostly written in Forth) usually fits in 6K bytes, running stand-alone with no other software required, or as a task in a conventional operating system. One person can understand the entire Forth system, change any part of it, or even write a new version from scratch. Run-time efficiencies are as little as 30% slower than straight machine code, and even less if the system's built-in assembler is used. When the assembler is not used, programs can be almost completely transportable between machines. Any large Forth program is really a special-purpose, application-oriented language, greatly facilitating maintenance and modification. We don't yet have conclusive data, but typical program development times and costs seem to be a fraction of those required by Fortran or assembly. Forth is especially useful for real-time, control-type applications, for large projects, and for small machines.

The problem is availability. Users have shown an ease of learning after they have a system available. The Forth characteristics of postfix notation, structured conditionals, and data stacks are best understood by use. To encourage Forth programmers, we need readily available systems even of modest performance. We hope that three levels will be available:

1. Demonstration - free (or under $20.) introductory version without file structure which compiles and executes from keyboard input.

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

2. Personal - low cost ($10. to $100.) with RAM or tape based files.

3. Professional - Commercial products for lab or industrial use and software development. ($1000. to $2500.)

Today the serious personal computer user holds the key to wider availability of the language. These users - generally engineers, businessmen, programmers - combine professional competence and commitment with the freedom to try new methods which may require a lot of time and tinkering with no definite guarantee of payoff. Practically everyone involved with the Forth Interest Group has both a personal and a professional interest in computers.

The Forth Interest Group is non-profit and non-commercial. We aren't associated with any vendor, no one is making money from it, and we are all busy with other work. We are an information clearinghouse and want to encourage distribution of all three of the previously mentioned levels of Forth. We do not have a Forth system for distribution at this time, and we don't want to get into the software or mail-order business because this is best left to companies or individuals committed to that goal. Naturally our critical issue is how to keep going over the long haul with volunteer energy. We need cost-effective means of information exchange.

At present we are writing for professional media, putting out this simple newsletter, and holding occasional meetings in the Bay Area. Also, we are developing a major technical and implementation manual, to be published in a journal form as four installments, available by subscription. While we cannot answer all of the mail individually, we certainly read it all, to answer it in the newsletter. While we cannot fill orders for software or literature, we will try and point you to where it is available. We welcome your input of information or suggestions, how you could help, what you would like to see happen, and where we should go from here.

Dave Bengel - Dave Boulton - Kim Harris - John James
Tom Olsen - Bill Ragsdale - Dave Wyland

EVOLUTION OF A F.I.G. FORTH FREAK

By Tom Olsen

I have been actively involved in the personal computing movement since early in 1974 when I shelled out $120. for an 8008 chip. Since that time my hardware and software have evolved into a very powerful and useful system, of which FORTH is a principal component. The system consists of an LSI-11, 28K of memory, 2 Diablo disks, an LA30 DECWRITER, a Diablo HYTYPE-I printer, a VDM-1 display, and dual floppy disk drives. Obtaining an operating system which would effectively utilize all of this hardware initially appeared to be much too expensive for an individual to buy, and far too complex to write from scratch. This attitude changed when early in 1976 I read a technical manual describing the internal organization of a relatively unknown "language" called FORTH. Here was a programming system which included not only an editor, assembler, and file management system, but the inherent capability to be rapidly expanded to perform any computer function I could define. The best part was the fact that the central core of this programming system was relatively small and would easily fit into 3K of memory. The large majority of the system programming could be done in terms of high level functions which I would have the freedom to define.

After about three months of late nights and pulling my hair out, I finally had a stand alone FORTH system which I could bootstrap and then use to load application vocabularies from disk. Once the basic implementation was fully debugged my general throughput of useful application software increased to a level I never would have thought possible. I can't over-emphasize the satisfaction associated with implementing the language from scratch. An added benefit of this approach is the flexibility derived by having a 100% understanding of ALL of the code your machine is executing.

Today I have application vocabularies which can do everything from playing a BACH minuet on a computer controlled synthesizer to generating, sorting, and printing the FORTH INTEREST GROUP mailing list. It is my hope that with the continued growth of the FORTH INTEREST GROUP and the establishment of some syntactical standards, widespread exchange of applications vocabularies will greatly enhance the computing power of all users of FORTH-like languages.

USING FORTH FOR TRADEOFFS BETWEEN HARDWARE/FIRMWARE/SOFTWARE

By Dave Wyland

FORTH provides a unique capability for changing the tradeoffs between software, firmware, and hardware. This capability derives from the method of nested definition of operators in FORTH.

Firmware (i.e. microprogramming) can add new instructions to the instruction set of an existing machine. New hardware designs can also add instructions to an existing set, with the Z80 upgrade of the 8080 serving as a good example. These new instructions could significantly improve the throughput of a system in many cases: however these new instructions cannot be used with existing software without rewriting the software. This is because current software methods such as assembler language, BASIC interpreters, FORTRAN compilers, etc., create software programs as one list of instructions. To add improved instructions to a program generated by a FORTRAN compiler involves rewriting the compiler to efficiently use the new instructions and then recompiling the program. This is not a trivial task since decisions must typically be made as to when to use a new instruction. This is why each new machine requires a new, rewritten FORTRAN compiler. The fact that the job has been done many times before is not very comforting.

With FORTH, however, operations are built-up of nested definitions with a common functional interface between operations. If a new instruction has been added to the computer's instruction set, it can be added to the FORTH system by changing a small number of definitions, in the typical case. The method can be quite straightforward. Each new instruction does an operation which would have required several instructions in the previous case. By identifying those FORTH definitions which could benefit from the new instructions and recoding them to include it, all software which uses the modified definitions is immediately improved. Also, very few definitions will have to be modified: only those elementry definitions with a high frequency of use need be modified to achieve throughput increase near the maximum possible.

Since the FORTH definitions are nested, the throughput gain of new instructions can be achieved by modifying a small amount of code, and the remaining structures remain unchanged. All existing application programs, translators, etc. are immediately improved without change!

Note that this process is reversable. Programs created for an existing system which has an extensive instruction set can be run on a simpler machine by converting the appropriate code based definitions to nested (colon) definitions. The process is also incremental: new instructions can be added one-at-a-time as desired.

FORTH LEARNS GERMAN

By John James

Forth now understands German, thanks to the following redefinitions of the operations in the Decus (Caltech) Forth manual.

Many of the operations were mathematical symbols, and of course they did not have to be translated. Of the rest, the control operations (IF, THEN, ELSE, BEGIN, END, DO, LOOP, +LOOP) are special, because they are "immediate operations"; that is, they are executed at compile time. Just redefining their names would not work, because they would try to execute right in the definitions. So their original definitions were copied, but with the German names.

Program development time for bilingual capability, two hours. Memory required, 600 bytes. Effect on run-time execution speed, zero.

```
BLOCK 30
 1 ( GERMAN.  JJ, 6/19/78)
 2 : ABWERFEN DROP ;     : UBER OVER ;
 3 : VERT SWAP ;    : SPAREN SAVE ;    : UNSPAREN UNSAVE ;
 4 : UNBEDINGT ABS ;     : UND AND ;
 5 : HOCHST MAX ;     : MINDEST MIN ;     : REST MOD ;
 6 : ODER OR ;    : /REST /MOD ;    : OSETZEN OSET ;
 7 : 1SETZEN 1SET ;    : HIER HERE ;    : VERGESSEN FORGET ;
 8 : SCHLUSSEL CODE ;    : BESTANDIG CONSTANT ;
 9 : GANZE INTEGER ; : ORDNUNG ARRAY ; : IORDNUNG IARRAY ;
10 : SETZEN SET ;
11 : AUFS-LAUFENDE UPDATE ; : AUSRADIEREN ERASE-CORE ;
12 : LADEN LOAD ;    : ZURUCK CR ;
13 : SCHREIBEN TYPE ;     : BASIS BASE ;
14
15
16 ( THE SAME: DUP, MINUS, =L, BLOCK, ;S, F)    ;S
```

The control operation definitions (OB, DANN, SONST, BEGINNEN, ENDEN, TUN, SCHLINGE, +SCHLINGE) are not shown here; they are all short (one line), and exact copies of the English operations. (Incidently this particular vocabulary is a rough draft; we have not seen the results of the International Forth Standards Team, which is currently at work.) The word GERMAN is defined on the load screen, so that the Forth user can call in the German vocabulary when desired. The following session shows the entry and execution of a fibonacci

sequence program in German (until 16-bit overflow).

```
FORTH LOAD
OK
GERMAN
OK
: PRUFUNG 0 1 30 0 TUN VERT DUP . UBER + SCHLINGE ABWERFEN ABWERFEN ;
OK
PRUFUNG
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 1771
 28657 -19168 9489 -9679 -190 -9869 -10059
```

On a (slightly) more speculative note, why not extend this scheme to a computer assisted international language for computer conferences, electronic mail, and international data-base utilities? Clearly natural language is too free, and computer languages like BNF are too restrictive, to be feasible. But a hybrid, a vocabulary of several hundred unambiguous words (each used in one sense only), and perhaps some computer-oriented syntactical markers, should be enough for useful dialog within a particular interest area. If it works for two languages it should work as well for any number. The final test - whether international teams could collaborate, after minimal training - would take a few weeks programming at most, after the vocabularies and terminal interfaces had been determined.

FORTH MAILBAG

By Dave Bengel

The Forth Interest Group developed from several people in the San Francisco- San Jose area who have been working on Forth for the last year and a half. Until about six months ago most of these users were unaware of each other. Until the publication of the article by John James in Dr. Dobb's Journal (May 1978), about 80 percent of the group was from the Homebrew Computer Club - whose publication should also be watched for news concerning the F.I.G.

We now have nearly 200 names on the mailing list, and are receiving about six letters a day. The writers' main question is how to get a version for their machine.

We don't yet have a detailed description of the versions of Forth now available, nor is there a standard form of the language available for various CPUs. Intense work on implementations is now underway; e.g. the Forth Interest Group implementation workshop. We will keep you informed as more documentation and systems become available.

Your answers to the questionaire in this newsletter will help us keep a mailing list for interest-specific applications, documentation, CPU versions, etc. We appreciate any information you can send us, particularly about Forth versions or variants which are running or being developed, or any software we can publish. We need your contributions to this newsletter (which we hope will grow into a journal).

FIG LEAVES

IMPLEMENTATION WORKSHOP

1771

The Forth Interest Group will hold an implementation workshop, starting sometime in July. The purpose is to create a uniform set of implementations for common micros. The assembly listings which result will be available through F.I.G. to those who wish to distribute specific versions.

This will be a small group, no more than ten, with only one person for each machine. There will be approximately four all-day sessions, over six weeks. Implementers must have access to their target machine, with an assembler and editor; floppy or tape is not required. The meetings will be to share notes and specific guidance on implementation details. At the end of the workshop we should have uniform Forth versions for all the machines.

We now have implementers scheduled for 8080, 6502, PACE, and LSI-11. We need implementers for 6800, Z80, 1802, F8, System 3 (32), 5100, etc. We also need a project librarian with Forth experience.

If you are interested write to FORTH IMPLEMENTATION PROJECT, 20956 Corsair Blvd., Hayward, Ca. 94545.

CONTRIBUTED MATERIAL

Forth Interest Group needs the following material:

(1) Manuals available for distribution. We can purchase copies and distribute, or print from your authorized original.

(2) Name and address of Forth implementations for inclusion in our publications. Include computer requirements, documentation, and cost.

(3) Technical material for the forthcoming journal. Both expositions on internal features of Forth and application programs are needed.

(4) Users who may be referenced for local demonstration to newcomers, on a regional basis. Indicate interest area (i.e. personal computing, educational, scientific, industrial, etc.).

(5) Letters for publication in this newsletter.

FORTH DIMENSIONS

AUGUST/SEPTEMBER 1978 VOLUME 1 NO. 2

CONTENTS

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

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 though 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

FOR NEWCOMERS

FORTH listings consist of sequences of "words" that execute and/or compile. When you have studied a glossary and a few sample listings, you should develop the ability to understand the action of new words in terms of their definition components. For the time being, we present a simplified glossary of the undefined words in this issue of FORTH DiMENSTIONS. For a fuller listing send for the F.I.G Glossary.

: xxx ..... ;
':' creates a new word named 'xxx' and compiles the following words (represented at ....) until reaching ';'. When 'xxx' is later used, it executes the words right after its name until the ';'.

CONSTANT VARIABLE
Each creates a new word with the following name, which takes its value from the number just before.

IF ELSE THEN
A test is made at 'IF'. If true, the words execute until the ' ELSE' and skip until THEN. If false, skip until ELSE and execute until THEN.

BEGIN END
At END a test is made; if false, execution returns to BEGIN; otherwise continue ahead.

DO LOOP LEAVE
At DO a limit and first index are saved. At LOOP, the index is incremented; until the limit is reached, execution returns to DO. LEAVE forces execution to exit at LOOP.

DUP DROP OVER SWAP ROT + - * /
These words operate on numbers in a stack just as then do in a HP calculator. If you like HP, you'll love FORTH.

>R R>
>R moves the top stack number to another stack. R> retreives it back to the original stack.

@ C@
@ Fetches the 16 bit contents of an address. C@ does the same for a byte. C@ may be also called B@ or \@.

! C!
These words store the second stack number at the memory address on the top of the stack. C! stores only a byte; it may be named B! or \! on some systems.

TYPE types a string by memory address and character count.

SPACE types a space.

CR types a carriage return/line feed.

MOVE moves within memory by addresses and byte count.

. prints a number.

.R prints a number in a tabulated column.

2+ adds two to the stack top number.

STATE is a variable, true when compiling.

(skips over comments until finding a ')'.

EXECUTE executes the word whose address is on the stack.

' finds the address of the next input name.

AND is a bitwise logical and.

, places a number in memory as part of compiling.

= > < are logical comparisons of stack numbers.

20 WORD fetches the next input word string.

HERE is a temporary memory workspace.

IMMEDIATE <BUILDS DOES> are too involved to discuss here. They are described in some detail in the text.

EDITORIAL

FORTH DIMENSIONS is dedicated to the promotion of extensible, threaded languages, primarily FORTH. Currently we are seeing a proliferation of similar languages. We will review all such implementations, referring to sources and availability.

Our policy is to use the developing "FORTH 77" International Standard as our benchmark.

Variant languages, such as STOIC, URTH, and CONVERS, will be evaluated on their advantages and disadvantages relative to FORTH. However, in evaluating languages named FORTH, we will note their accuracy in implementing all FORTH features. We expect complete versions named FORTH to contain:

1. indirect threaded code
2. an inner and outer interpreter
3. standard names for the 40 major primitives
4. words such as ;CODE, BLOCK, DOES>, (or ;:), which allow increased performance.

We hope to enable prospective users/purchasers to correctly select the version and performance level they wish, to foster long-range growth in the application of FORTH.

W.F.R.

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.

EXTENSIBILITY WITH FORTH

The purpose of any computer language (and its compiler or interpreter) is to bridge the gap between the "language" the machine understands (low level) and a language people understand (high level programming language). There are many choices for human-understandable languages: natural languages and artificial languages. The choice of language should allow convienient, terse, and unambiguous specification of the problem to be solved by the computer. Ordinarily only a few computer languages are available (e.g. BASIC, FORTRAN, APL). These were designed for certain classes of problems (such as mathematical equations) but are not suitable for others. The level of a language is a measure of suitability of that language for a particular application. The higher the level, the terser the program. By definition, [1] the highest level would allow a given problem to be solved with one operator (or command) and as many operands as there are input data required.

A natural language (e.g. English) might appear to be the best choice for a human-understandable computer language, and for some applications it may be. But natural languages suffer from three limitations: verbosity, ambiguity, and difficulty to decipher. This is partly because the meaning of a given word is dependent on its useage in one or more sentences (called "context sensitive") and because they require complex and nonuniform grammar roles with many exceptions. Specialized vocabularies and grammars permit terse and precise expression of concepts for restricted sets of problems. For example, [2] consider the following definition of a syllogism from propositional calculus:

$$((P1 \supset P2) \supset ((P2 \supset P3) \supset (P1 \supset P3)))$$

This sentence may be translated into English as "Given three statements which are true or false, if the truth of the first implies the truth of the second, this implies that if the truth of the second implies the truth of the third, then the truth of the first implies the truth of the third." Ambiguity is hard to avoid in most natural languages. The English phrase "pretty little girls school" (when unpunctuated) has 17 possible interpretations! (Try it.) [3]

As for the suitability of traditional programming languages (e.g. BASIC, FORTRAN, COBOL, PASCAL, APL) for "almost all technical problems", try coding the following "sentences" in your favorite computer language:

Quantum Mechanics:

$$H\psi = E\psi$$
where $H = -(\hbar^2/2m)\nabla^2 + V$
and E is the energy of the system

Electricity and Magnetism:

$$\nabla \cdot D = \rho$$
$$\nabla \cdot B = 0$$
$$\nabla \times E = -\frac{\partial B}{\partial t}$$
$$\nabla \times H = i + \frac{\partial D}{\partial t}$$

Matrix Algebra:

The trace of a matrix is equal to the sum of its eigenvalues.

Organic Chemistry:

$$3CH_3CH_2OH + Cr_2O_7^{--} + 8H^+ \rightarrow 3CH_3CHO(g) + 2Cr^{+++} + 7H_2O$$

Knitting:

K2 tog.3(5) times, *k1, p2, k1, k3, tog., k1,
p2, k1, p1, k1, p3 tog., k1, p1, p1* ; repeat
between *'s once more, k1, p2, k1, k3 tog.,
k1, p2, k1; k2 tog. 3(5) times; 47(51) sts.

Poetry:

Shakespearean sonnets are in iambic pentameter and
consist of three quatrains followed by a couplet.

FORTH is capable of being matched to each of the above relations at a high level. Furthermore, using the FORTH concept of vocabularies, several different applications can be resident simultaneously but the scope of reference of component words can be restricted (i.e., not global). This versatility is because the FORTH language is extensible. In fact the normal act of programming in FORTH (i.e., defining new words in terms of existing words) extends the language! For each problem programmed in FORTH, the language is extended as required by the special needs of that problem. The final word defined which solves the whole problem is both an operator within the FORTH language (which is also a "command") and the highest level operator for that problem. Further, the lower level words defined for this problem will frequently be useable for the programming of related problems.

It is true that popular computer languages allow new functions to be added using SUBROUTINES and FORTRAN-like FUNCTIONS. However these cannot be used syntactically the same as the operators in the language.

For example, let's assume a BASIC interpreter does not have the logical AND operator. To be consistent with similar, existing operators one would like to use "AND" in the following syntax:

A AND B

(A,B are any valid operands). Furthermore, one would want this new operator to have a priority higher than "OR" and lower than "NOT" so that

NOT A AND B OR C

would mean

(NOT A) AND (B OR C)

The only way to do this is to modify one's BASIC interpreter. Although not impossible, this is usually very difficult because of the following reasons:

(1) Most BASIC's are distributed without the interpreter source code (or not in machine-readable form). (2) One would have to learn this program and design a change. One modification might be "patched in", but many probably could not. Such a modified interpreter might not be compatible with future releases from the manufacturer. Errors might be introduced into other parts of the interpreter by this modification. (3) The process of changing is time consuming. Before one could try the new version, one would have to assemble, link, load, and possibly write a PROM. (How long would this take on your system?)

Another alternative would be to use a BASIC FUNCTION to add the AND operator, but it would have to be referenced as:

AND (A,B)

If NOT and OR were also FUNCTIONS, (NOT A) AND (B OR C) would have to be written as:

AND (NOT(A), OR(B,C))

This later form is lower level, less readable, and inconsistent with the intrinsic operators.

The addition of an AND operator in FORTH is as simple as any other programming addition; it would require one line of source code. The FORTH assembler could be used to take advantage of a particular processor's instruction, or the compiler could be used, resulting in machine transportability! Other differences from the example of modifying BASIC are: (1) Nothing existing in the FORTH system needs to be changed, so no learning is required, no errors are introduced to the existing system, and compatibility with future releases is preserved. The source of the FORTH kernel is not even necessary. (2) The new operator can be tried immediately after it is defined; if it is wrong it can be fixed before any further use is made of it. (3) This new operator is used exactly the same as any FORTH operator. So it may be mixed with existing operators in a totally consistant manner.

APL fans will point out that all the above is true for APL also. For something as simple as "AND" there is little difference. However, APL allows only monadic (single operand) and diadic (two operand) operators; FORTH operators can be written to accept as many operands as the programmer desires.

The previous discussion addressed the extension of the FORTH language, but it is almost as easy to extend the FORTH compiler! New compiler control structures (e.g., the CASE construct) can be added without changing any of the existing compiler. Or the existing compiler can be modified to do something different. To maintain compatibility with the existing compiler, the modifications could be part of a user- defined "vocabulary" so that both versions would be selectively available. Furthermore, one can write an entirely new compiler which accepts either the FORTH language or another language. (Complete BASIC interpreters have been written in FORTH.)

The choice of a computer language for a given application (including system development) should optimize the following attributes: (1) Be terse (i.e., the highest level for the application) (2) Be unambiguous (3) Be extensible (e.g., language, data types, compiler) (4) Be efficient (5) Be understandable (e.g., self documentation) (6) Be correct (e.g.,testing, proving assertions, consistency checks) (7) Be structured (e.g., structured programming, reentrant, recursive) (8) Be maintainable (e.g., modular, no side effects)

FORTH is a compromise among these goals, but comes closer than most existing programming languages.

;S KIM HARRIS

REFERENCES

1 Halstead, Maurice, Language Level: A Missing Concept in Information Theory, Performance Evaluation Review, ACM SIGMETRICS, Vol. 2 March '73.

2 McKeeman, Horning, and Wortman, A Compiler Generator, Prentice-Hall, 1970.

3 Brown, James Cooke, Loglan 1: A Logical Language, Loglan Institude, 2261 Soledad Rancho Road, San Diego, CA (714) 270-9773

GERMAN REVISITED

In the last issue of FORTH DIMENSIONS we showed how to create a bi-lingual (or multi-lingual) version of FORTH, and listed a simple program (set of FORTH definitions) for doing so. In respect to translation, there are three different classes of FORTH words:

(A) Those such as mathematical symbols which don't need to be translated.

(B) Words such as DO and IF which cannot be translated by a simple colon definition; the existing definitions must be re-copied and given German names. (all the definitions are short - one line - however)

(C) Other words, which could either be re-copied, or re-defined by a colon definition.

In any case, separate vocabularies can be used to prevent spelling clashes, no matter how many languages are spoken by one FORTH system. It can be possible to change languages as much as desired, even in the middle of a line.

The article stated that there was no run-time overhead. Such performance is possible, but the example given does have a run-time overhead of one extra level of nesting for each use of a word translated by a colon definition.

The following article by Bill Ragsdale is a more advanced treatment of language translation methods. It is written at the level of the FORTH systems programmer, and it uses a more standard FORTH version than the DECUS-supplied version which was used in the article which appeared in FORTH DIMENSIONS 1.

JOHN S. JAMES

FORTH LEARNS GERMAN

In the last issue of FORTH DIMENSIONS, we featured an article on natural language name conversions for FORTH. This article will add some additional ideas on the same topic.

First, the method shown (vectoring thru code) does have some run-time overhead. Also, some code definitions cannot execute properly when vectored in this manner, for example:

```
: R>    R>   ;
```

will pull the call of R> from the return stack and crash. We would ultimately like to translate names with:

1. Precisely correct operation during execution and compiling.

2. A minimum of memory cost.

3. A minimum of run-time cost.

4. A minimum of compile-time cost.

Let us now look at three specific examples to further clarify some of the trade-offs involved.

EXAMPLE 1 - COMPILING WORD

Let us see how UBER can be created to self-compile.

```
HEX
: D-E >R 2+ @ (2+ optional on some systems)
 STATE @ IF  (compiling) DUP 8 - C@ 80 <
  IF 2 - , ELSE (immediate ) EXECUTE THEN
 ELSE EXECUTE
  THEN ;
 : DO ENGLISH EMPLACE D-E ' , IMMEDIATE ;
   IMMEDIATE
 : UBER DO.ENGLISH OVER ;
 : LADEN DO.ENGLISH LOAD ; etc.
```

When building the translation vocabulary, the colon ':' creates the word UBER and then executes the immediate word DO.ENGLISH. DO.ENGLISH first emplaces the run-time procedure 'D-E' and then uses " ' , " to emplace the parameter field address of the next source word (OVER). Finally, the new word (UBER) is marked immediate, so that it will execute whenever later encountered.

Now we see how UBER executes. When it is interpreted from the terminal keyboard, 'D-E' will execute to fetch the emplaced PFA within the definitions of UBER (by R> 2+ @). After checking STATE the ELSE part will execute OVER from its parameter field address.

When UBER is encountered by the compiler in a colon definition, it will execute, as do all compiling words. Again R> 2+ @ will fetch the PFA of OVER to the stack. The check of STATE will be true and DUP 8 - C@ will fetch the byte containing the precedence bit. When compared to hex 80, a true will result for non-immediate, and 2 - , will compile the code field address.

However, if the word had been immediate (OVER isn't) the ELSE part will execute the word as in any compiling word.

The space cost of example 2 is 14 bytes per word (8 bytes per header and 6 bytes in the parameter field). The compile time cost is the execution of 'D-E.' There is no ultimate run-time cost in compiled definitions.

EXAMPLE TWO - <BUILDS - DOES>

Another way is to define a 'BUILDS-DOES' word E>G (English to German). It is then used to build a set of translation words similar to a FORTH mnemonic assembler.

```
: E>G   <BUILDS  ' , IMMEDIATE
        DOES>  @  STATE  @
        IF (compiling )  DUP 8 - C@ 80 <
            IF 2 - , ELSE EXECUTE THEN
          ELSE  EXECUTE  THEN  ;

E>G  UBER  OVER
E>G  LADEN  LOAD
E>G  BASIS  BASE     ..... etc.
```

E>G is a defining word that builds each German word (UBER) and emplaces the parameter field address of the English word (OVER) into the new parameter field (of UBER), and finally makes UBER immediate. When UBER is encountered by the outer interpreter, it does the DOES> part. The parameter of UBER, (the PFA of OVER) will be fetched and STATE tested. Since executing, the ELSE part will execute OVER from its parameter field address (as in the example 1).

When compiling, the DOES> part will be executed, again similarly to 'D-E' as in Example 1. The space cost of the BUILDS-DOES method is 10 bytes per word (8 in the header, 2 in the parameter field). The compile time is the same as in Example 1 and there is no run-time cost.

EXAMPLE THREE - RENAME

This last method is the most fool-proof of all. We will just re-label the name field of each resident word to the German equivalent.

```
: RENAME  '  8  -  DUP  C@  80

   AND  (precedence  bit   )
   20  WORD  HERE  C@  +
   HERE  C!  (store  into length)
   HERE SWAP 4 MOVE ;

     ( overlay old name )

   RENAME OVER UBER

   RENAME LOAD LADED

   RENAME BASE BASIS
```

This method extracts the precedence bit of the old (English) definition and adds it to the length count of the new (German) name. The new name is then overwritten to the old name field. There is no space or time cost!! The dictionary is now truely translated.

A final caution is in order for Examples 1 and 2. Some FORTH methods may still give trouble. If you should try:

```
UBER
```

you will find the PFA of UBER which is a translating definition, and not the ultimate run-time procedure, (which is really in OVER). This would have disasterous results if you were attempting to alter what you thought was the executing procedure, and you were really altering the compiling word. For this reason, the method of Example 3 is the only truly 'fool-proof' method. The renaming method has the added use of allowing you to change names in your running system. For example, it is likely that the old <R will be renamed >R in the international standard FORTH-77. You can simply update your system by the use of the word RENAME.

;S W.F. RAGSDALE 8/27/78

THREADED CODE

Bell (1) and Dewar (2) have described the concepts of Threaded Code (also called Direct Threaded Code, or DTC), and an improvement called Indirect Threaded Code, or ITC. DTC was used to implement Fortran IV for the PDP-11, and ITC was used for a machine- independent version of Spitbol (a fast form of the string-processing language Snobol). Forth is a form of ITC, but different from the scheme presented in (2).

In DTC, a program consists of a list of addresses of routines. DTC is fast; in fact, only a single PDP-11 instruction execution is required to link from one routine to the next (the instruction is 'JMP @(R)+', where 'R' is one of the general registers). Overall, DTC was found to be about three percent slower than straight code using frequent subroutine jumps and returns, and to require 10-20 percent less memory. But one problem is that for each variable the compiler had to generate two short routines to push and pop that variable on the internal run-time stack.

In ITC, a program is a list of addresses of addresses of routines to be executed. As used in (2), each variable had pointers to push and pop routines, followed by its value. The major advantage over DTC is that the compiler does not have to generate separate push and pop routines for each variable; instead these were standard library routines. The compiler did not generate any routines, only addresses, so it was more machine independent. In practice, ITC was found to run faster than DTC despite the extra level of indirection. It also used less memory.

Forth is a form of ITC, with additional features.

Forth operations are lists of addresses pointing into dictionary entries. Each dictionary entry contains:

(A) Ascii operation name, length of the name, and precedence bit; these are used only at compile time and will not be discussed further.

(B) A link pointer to the previous dictionary entry. (This is used only at compile time.)

(C) A pointer called the code address, which always points to executable machine code.

(D) A parameter field, which can contain machine instructions, or Forth address lists, or variable values or pointers or other information depending on the variable type.

By the way, virtually everything in Forth is part of a dictionary entry: the compiler, the run-time routines, the operating system, and your programs. In most versions, only a few bytes of code are outside of the dictionary.

The code address is crucial; this is the 'indirect' part of ITC. Every dictionary entry contains exactly one code address. If the dictionary entry is for a "primitive" (one of the 40 or so operations defined in machine language), the code address points two bytes beyond itself, to the parameter field, which contains the machine-language routine.

If the dictionary entry is for a Forth higher-level operation (a colon definition), the code address points to a special "code routine" for colon definitions. This short routine (e.g. 3 PDP-11 instructions) nests one level of Forth execution, pushing the current 'I' register (the Forth "instruction counter") onto a return-address stack, then beginning Forth execution of the address-list in the new operation's parameter field.

If the dictionary entry is for a variable, then the code address points to a code routine unique to that variable's type. The parameter field of a variable may contain the variable's value - or pointers if re-entrant, pure-code Forth is desired.

Results of Forth-type ITC include:

(A) Execution is fast, e.g. two PDP-11 instruction executions to transfer between primitives, about ten to nest and un-nest a higher-level definition. (Because of the pyramidal tree-structure of execution, the higher-level nesting is done less often.) Yet the language is fully interactive.

(B) Forth operation names (addresses) are used exactly the same regardless of whether they represent primitives or higher-level definitions (nested to any depth). Not even the compiler knows the difference. In case run-speed optimization is desired, critical higher-level operations (such as inner loops) can be re-coded as primitives, running at full machine speed, and nothing else need by changed.

(C) Forth code is very compact. The language implements an entire operating system which can run stand-alone, including the Forth compiler, optional assembler, editor, and run-time system, in about 6k bytes. (Forth can also run as a task under a conventional operating system, which sees

it as an ordinary assembly-language program, and Forth can link to other languages this way.) Code is so compact that application-oriented utility routines can be left in the system permanently, where they are immediately available either as keyboard commands or instructions in programs, and they are used in exactly the same way in either case. No linkage editing is needed, and overlays are unusual.

REFERENCES

(1) Bell, James R. Threaded code. C. ACM 16, 5 (June 1973), 370-372.
(2) Dewar, Robert B. K. Indirect threaded code. C. ACM 18, 6 (June 1975), 330-331.

STAFF

The volunteer staffing of FORTH DIMENSIONS is a bit fluid. For this issue, our staff consisted of :

EDITOR	JOHN JAMES
CONTRIBUTORS	KIM HARRIS W.F.RAGSDALE
TYPESETTING	TOM OLSEN JOHN JAMES
ARTWORK	ANNE RAGSDALE
CIRCULATION	DAVE BENGEL
DATA PROCESSING	P D P -11

FORTH DEFINITION

FORTH is the combination of an extensible programming language and interactive operating system. It forms a consistent and complete programming environment which is then extended for each application situation.

FORTH is structured to be interpreted from indirect, threaded code. This code consists of sequences of machine independent compiled parameters, each headed by a pointer to executable machine code. The user creates his own application procedures (called 'words'), from any of the existing words and/or machine assembly language. New classes of data structures or procedures may be created; these have associated interpretive aids defined in either machine code or high level form.

The user has access to a computation stack with reverse Polish conventions. Another stack is available, usually for execution control. In an interactive environment, each word contains a symbolic identifier aiding text interpretation. The user may execute or compile source text from the terminal keyboard or mass storage device. Resident words are provided for editing and accessing the data stored on mass storage devices (disk, tape).

In applications that are to run 'stand-alone', a compact cross-compiled form is used. It consists of compiled words, interpretive aids, and machine code procedures. It is non-extensible, as the symbolic identifiers are deleted from each word, and little of the usual operating system need be included.

;S W.F.R. 8/26/78

NOTES

- The second meeting of the FORTH International Standards Team will occur in Los Angeles on October 16-19. Contact FORTH Inc. for additional information.

- A partially micro-coded FORTH-like language is described in "Threaded Code for Laboratory Computers" by J.B. Phillips, M.F. Burke, and G.S. Wilson, Dept. of Chemistry, University of Arizona, Tucson, AZ 85721. The article is published in Software - Practice and Experience, Volume 8, pages 257-263. Implementation is on a HP2100. The article also describes the advantages of threaded languages for laboratory applications.

- A "form" of FORTH for the Apple and PET 6502 based computers is available from Programma Consultants, 3400 Wilshire Blvd., Los Angeles, CA 90010. We have not used these enough to review them for this issue but they have been shipped and do work. For more information write to Programma Consultants or watch future issues of FORTH DIMENSIONS.

- FORTH Inc. is looking for a programmer with some systems-level experience using FORTH or similar languages. Interested persons should contact FORTH Inc., 815 Manhattan Avenue, Manhattan Beach, California 90266, (213) 372-8493 .

HELP

```
SCR # 6

 0      THE 'HELP' COMMAND IS PROBABLY THE MOST USEFUL OPTION FOR
 1 A FORTH SYSTEM.  IT ALLOWS YOU TO VIEW THE DICTIONARY WORDS
 2 AND LOCATE THEM IN MEMORY.  WHEN YOU ARE TESTING NEW
 3 DEFINITIONS, IT WILL SHOW RE-DEFINITIONS.  IT IS A WAY TO
 4 LOCATE WHERE A MISSING WORD SHOULD BE, BUT ISN'T.
 5
 6      IF YOU MAKE A COMPILE ERROR FROM DISC. 'HELP' WILL SHOW
 7 THE WORD IN WHICH THE ERROR OCCURED.
 8
 9      YOU SHOULD MODIFY THE FOLLOWING DEFINITIONS TO THE FORMAT
10 YOU WANT.  FOR OBJECT CODE EXAMINATION, I LIKE THE CODE FIELD
11 ADDRESSES AS SHOWN, SINCE THIS IS WHAT RESULTS IN THE COMPILED
12 CODE.   FOR A QUICK SNAP-SHOT OF THE DICTIONARY, I JUST PRINT
13 THE LENGTH AND NAMES.
14
15      JUST TYPE 'HELP' AND HIT THE 'BREAK' KEY TO STOP.


SCR # 7

 0 (  HELP  )                HEX
 1 00      CONSTANT  LAST.LINK    ( IS $8000 ON MICRO-FORTH )
 2  4      CONSTANT  #/LINE       ( WORDS PRINTED PER LINE  )
 3
 4 : .NAME              ( ENTER WITH ADDRESS OF LENGTH BYTE )
 5    DUP  C@  7F  AND  DECIMAL  3  .R  SPACE  1+  3  TYPE SPACE ;
 6
 7 : .CODE-ADDRESS      ( ENTER WITH ADDRESS OF LENGTH BYTE )
 8      6  +  HEX  5  .R  SPACE ;
 9
10 : .HEADER            ( ENTER WITH ADDRESS OF LENGTH BYTE )
11      DUP  .NAME  .CODE-ADDRESS ;
12
13 : ?TERMINAL  0 ;  ( USER'S MACHINE DEPENDENT TERMINAL BREAK )
14              ( RETURN '00' FOR NO BREAK, AND '01' FOR A BREAK )
15 8 LOAD      ;S    8/27/78     WFR


SCR # 8

 0 (  HELP, CONT.  )
 1
 2 : .LINE              ( PRINT A LINE OF NAMES AND CODE ADDRESSES )
 3    #/LINE  0         ( ENTER WITH ADDRESS OF LENGTH BYTE        )
 4    DO  DUP  .HEADER  SPACE    4  +  @  DUP  LAST.LINK  =
 5        IF  LEAVE  THEN  LOOP ;     ( EXIT WITH NEXT ADDRESS )
 6
 7 : HELP          ( PRINT DICTIONARY FROM TOP CURRENT WORD DOWN )
 8           ( TO BOTTOM.  FORMAT IS LENGTH COUNT, 3 LETTERS OF )
 9           ( NAME, AND CODE FIELD ADDRESS.  WILL TERMINATE    )
10           ( UPON LAST LINK VALUE OR A TERMINAL BREAK.        )
11    BASE  C@  >R     CURRENT  @  @
12    BEGIN  CR  .LINE  DUP  LAST.LINK  =  ?TERMINAL  +
13       END    DROP ( LAST LINK )  R>  BASE  C!  ;
14
15 DECIMAL      ;S     8/28/78     WFR
```

```
HELP
  4 HEL   1EBB     5 .LI   1E90     7 .HE   1E80    13 .CO   1E66
  5 .NA   1E43     6 #/L   1E39     9 LAS   1E2F     4 TAS   1E25
  4 BOO   1B8B     2 ->    1B6A     4 TUB   1B57     3 TTY   1B44  OK
```

ABOVE WE SEE AN EXAMPLE OF THE LOADING OF THE 'HELP' COMMAND FROM DISC. IT THEN IS TESTED, AND DUMPS THE DICTIONARY. WE SEE THE LISTING OF 'HELP' AND THE WORDS IS USES LISTING CONTINUES INTO THE RESIDENT DICTIONARY.

GOOD LUCK, WFR

MANUALS

DECUS PDP-11 FORTH
by Owens Valley Radio Observatory, California Institute of Technology, Martin S. Ewing. (alias The Caltech FORTH Manual) Available from DECUS, 129 Parker Street, PK3/E55, Maynard, Mass. 01754. Ordering information: Program No. 11-232, Write-up, $5.00 .

FORTH Systems Reference Manual
W. Richard Stevens, Sep 76. Kitt Peak National Observatory, Tucson, AZ 85726. (NOT FOR SALE)

LABFORTH
An Interactive Language for Laboratory Computing, Introductory Principles, Laboratory Software Systems, Inc., 3634 Mandeville Canyon, Los Angeles, CA 90049. $8.00 .

STOIC
(Stack Oriented Interactive Compiler) by MIT and Harvard Biomedical Engineering Center. Documentation and listings for 8080 from CP/M Users Group, 164 west 83rd Street, New York, N.Y. 10024. $4.00 membership, $8.00 per 8" floppy, 2 floppies needed.

CONVERS
The Digital Group, Box 6528, Denver, CO 80226 Manual: DOC-CONVERS $12.50 .

URTH
(University of Rochester FORTH), Tutorial Manual, Hardwick Forsley, Laboratory for Laser Energetics, 250 E. River Rd., Rochester, NY 14621 .

microFORTH Primer
FORTH, Inc. 815 Manhattan Ave., Manhattan Beach, CA 90266 (moving soon) $15.00 .

(Page 21, 22 Blank)

FORTH DIMENSIONS

OCTOBER/NOVEMBER 1978 VOLUME 1, NO. 3

CONTENTS

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

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 though 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).

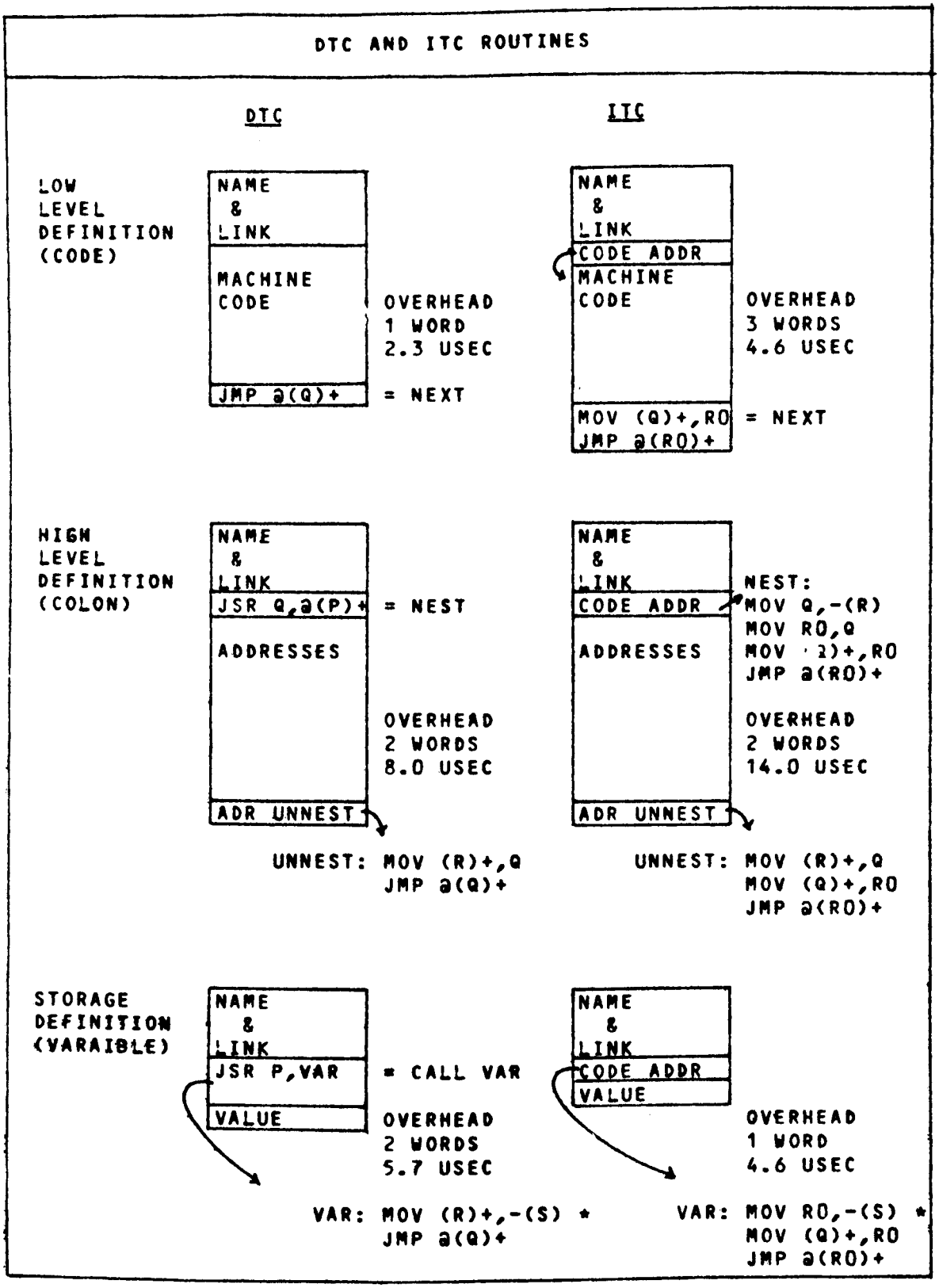
R is the return stack pointer (R6).

P is the program counter (R7).

R0 is a temporary register assumed to be available.

-->

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

	DTC	ITC	DTC ADVANTAGE
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

	SIZE ADVANTAGE	SPEED ADVANTAGE
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	.7 words	2.4 usec

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 independance because no machine code appears in the code generated by the compiler. But even this advantage is illusionary 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
David J. Sirag
Laboratory Software Systems, Inc.

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

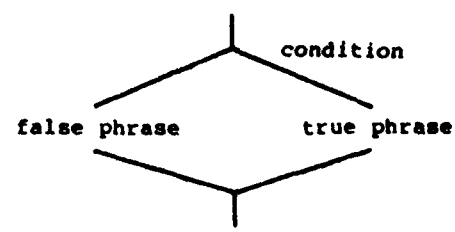
In FORTH, this structure takes the form:

```
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:

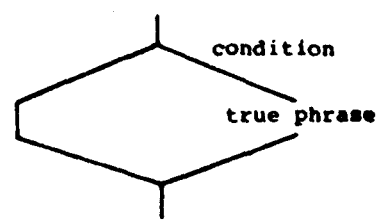


The "condition" is evaluated. If it is true, the "true phrase" is executed; otherwise, the "false phrase" is executed. The words following ENDIF (or THEN) are unconditionally executed.

If either phrase is omitted, as with

```
condition IF   true phrase   THEN
```

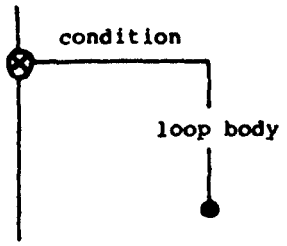
a vertical line is drawn as shown:



-->

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.

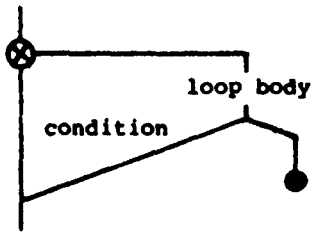
3. The "condition" is again evaluated. Its outcome might have changed during the execution of the loop statement.

   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   .
```

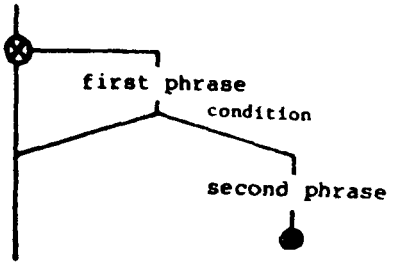
The loop body is always executed once, and is repeated as long as condition is false. The D-chart symbol for this structure would be:



A more general case is

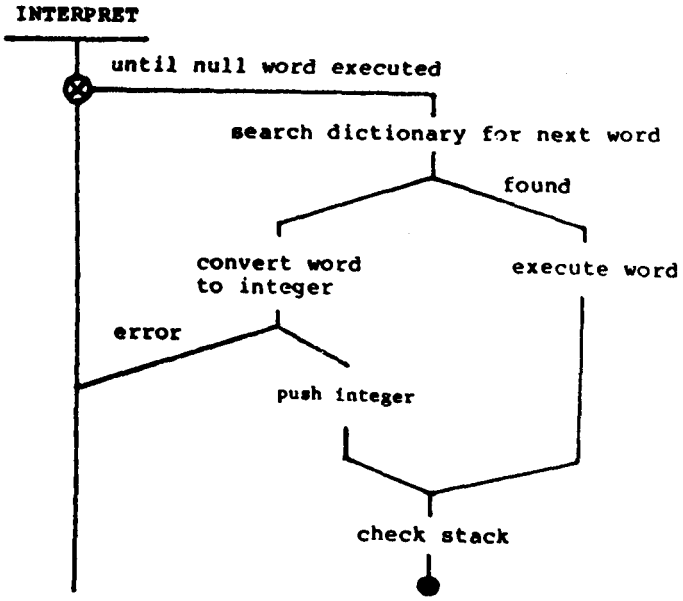
```
          BEGIN   first phrase
condition IF   second phrase
          AGAIN
```

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 F.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:

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 form 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 "PARN" 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

SYSTEM LANGUAGE 1

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 is 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 exeuction 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 mneumonic 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 Ø 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,Ø        ;stack in 16-bit
        JMP HPUSH      ;format
```

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

```
4ØØØ DB ØF ØF D2 ØØ 4Ø DB EC 6F
     26 ØØ C3 41 ØØ
```

This is exactly what the Assembly code would produce if ORG'ed at 4ØØØH 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 hign 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 Ø DO KEY 7F AND DUP ØD =
             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 Ø 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 Ø           ;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 ØD          ;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 OPCODE 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 RBM

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 disking 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 echo'ed to CRT to signal completion.

;s RBM Oct. 1978

FORTH DIMENSIONS

DECEMBER 1978/JANUARY 1979 VOLUME 1 No. 4
(Published July 1979)

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

PUBLIC MEETINGS OF F.I.G

The Forth Interest Group is pleased to announce a public meeting series. We will meet on the fourth (!) Saturday of the month in Hayward, Ca., in the Special Events Room of the Liberty House Department Store, in the Southland Shopping Center (800 Southland Mall)

This room is on the third floor rear. The formal meeting begins at 1:00 PM, but we gather for lunch about 12 Noon.

The specific dates are July 28, Aug 25, Sept 22, Oct 27, and Nov 24. A single technical topic will be presented in detail, with workshop sessions on the fig-FORTH model, and any topics of immediate interest. In July, Dave Lyons will present the ascii memory dump of Forth which is part of his 6800 version. This Forth program shows the entire language map on one sheet of paper!

PUBLISHERS COMMENTS

The issues of Forth Dimensions have been scheduled for two month intervals, but have been at intervals determined by the overall activities of the steering committee. In the past months we have participated in the International Standards Team, given conference papers the Fourth West Coast Computer Faire, attended the Forth Users Meeting in Utrecht, Holland.

These events have been outwardly reflected in the large gap since Issue 3. For the near term, hope is in sight!. We have Issue 5 ready for publication in three weeks, and Issue 6 should occur within a month. This will wrap up Volume 1. At this time we will evaluate our efforts, and plan for future activities. Member comment will be quite apropos during this period and will help shape future application of effort.

Thanks are due the membership for their patience during this period.

STAFF

The volunteer staffing of Forth Dimensions is a bit fluid. For this issue, our staff consisted of:

EDITOR	Bill Ragsdale
REVIEW	Dave Boulton
CONTRIBUTORS	Paul Bartholdi D. W. Borden
TYPESETTING	GLG Secretarial and Vydec
ARTWORK	Anne Ragsdale
CIRCULATION	6502 TIM and Persci

HISTORICAL PERSPECIVE

FORTH was created by Mr. Charles H. Moore in about 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of this 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 vendors.

The Forth Interest Group is centered in Northern California, although our membership of 450 is world-wide. It was formed in 1978 by local Forth programmers to encourage use of the language by the interchange of ideas through seminars and publications. All effort is on a volunteer basis and the group is affiliated with no vendors.

;S FIG

THE "TO" SOLUTION

Paul Bartholdi
Observatoire De Geneve
CH-1290-Sauverny
Switzerland

At the Catalina standardization meeting, Chuck Moore suggested rapidly the "TO" construct to aleviate some, if not all, of the difficulties associated with address manipulation. This new construction seems to me extremely powerful. It should considerably decrease the number of errors and improve the readability of programs. It is very easy to understand and to use (much easier in fact than the constructs it is replacing!). Its implementation is also trivial but the lack of experience in using it may hide some difficulties. These notes try to sketch its potentials.

1. The " TO " concept

The basic concept is the following: The code associated with VARIABLEs is divided into two exclusive parts.

- The first one (or " fetch code ") is identical with the one associated normally with CONSTANT . It pushes on the stack the value (byte, word or words) in the parameter field.

- The second one (or " store code ") is new. It transfers the value (byte, word or words) from the stack into the parameter field, it is equivalent to <variable-name> ! (or C! or D! or F!).

The choice between the two codes depends on a state variable (which will be called %VAR hereafter) that has two possible states: ø (or fetch state) and 1 (or store state).

If %VAR = ø the first code is executed and %VAR is unchanged.

If %VAR = 1 the second code is executed and % VAR is immediately returned to ø.

The operator " TO " sets %VAR to 1, forcing the next-called variable to store the content on top of the stack in its parameter field instead of fetching it.

2. Examples using " TO "

We suppose three single variables called A , B and C three double variables F , G and H , and three floating variables X , Y and Z . Then the following half-lines are equivalent. The first column assumes the old definitions of variables, the second one uses the new concept.

	OLD :	NEW :
i)	A @ B @ + C !	A B + TO C
ii)	F D@ G D@ D- H D!	F G D- TO H
iii)	X F@ Y F@ F* Z F!	X Y F* TO Z

3. VARIABLEs or CONSTANTs ?

Some advantages of the new concept are quite evident from the previous examples. Just before the Catalina meeting, I came to the conclusion that variables should be dropped altogether. If A , B , ... Z had been constants in the previous examples then, using P and ! , the three lines would have been coded

```
 i)  A  B  +  '  C  !
ii)  F  G  D- '  H  D!
iii) X  Y  F* '  Z  F!
```

which is already better than the "old" above.

The difficulty starts with ARRAYs. If they push values onto the stack instead of addresses, then it becomes much more difficult to store values at the right places.

Note that ' C (or H or Z) takes two words but is really, at execution time, a single operator (LIT <address-of-C>). " TO " is then the shortest solution in terms of memory space, and more or less equivalent to the CONSTANT solution in terms of the time used.

4. The " address " problem

But the main advantage of " TO " in this context are the following:

- In the general sense a " CONSTANT " should be used as such, and never (or hardly ever) be changed. In particular it may reside in PROM . I suggest then to keep the constants as such, its associated code ignoring the value of %VAR . We then should redefine the variables to check %VAR and behave accordingly.

- One of the main unresolved points at Catalina was the definition and use of addresses for variables in the general sense. One consensus was obtained in September at the Geneva meeting, that is, as far as possible, addresses should be omitted. The " TO " concept solves this requirement admirably. No addresses are

ever put on the stack, or manipulated explicitly. Then byte or word addressing is irrelevant. It is taken care of at the system level only, in the code of the second part (the "store code").

5. Portability

Because of this, FORTH programs become more portable. " TO " replaces all fetch and store operators which would or would not be distinct, which would work on byte or "position" or word addresses. Transporting a program from a micro to a large CDC implies now much less adaptations.

6. Clarity - security

Using less operators, in fact the strict minimum, is probably one of the best ways of improving clarity. Note also that " TO " appears as an infix operator to the programmer (and reader!). In terms of security, " TO " implies that only the parameter field of variables can be changed. Other addresses are not (at least directly) accessible.

This is certainly a tremendous gain for some environments.

7. The ARRAY problem

The use of " TO " with ARRAYs is not as simple as with variables, but still quite practicable. Note first that anything but a variable can stay between " TO " and the variable's name. If C is defined as an ARRAY (with the double associated code) then

```
A TO 4 C (instead of A @ 4 C ! )
DO I TO I C LOOP
(instead of DO I I C ! LOOP )
```

are quite alright and extend all the previously noted advantages of " TO ". But the programmer must take care that the index must not contain a variable. For example:

```
4 TO B ... A TO B C
```

will put the value of A into B instead of into C The necessary form should then be

```
A B TO C
```

which is surely less pleasant because of its asymmetry.

8. Fetching and storing inside a disk block

The problem extends of course to "virtual arrays" like the disk blocks. In this context, direct access should be considered separately from Data Management. The double code concept associated with " TO " should be extended to the Data Management operators. For the direct manipulation of data inside a disk block, I suggest the creation of a new operator, with the double code, associated with the form of

```
<relative-word-address>
<bloc-number> BLOC
```

(BLOC is of course just a provisional name!)

Example:

```
4 12 BLOC 5 + TO 5 12 BLOC
```

instead of

```
12 BLOCK DUP 4 + @ 5 +
SWAP 5 + !
```

or

```
12 BLOCK 4 + @ 5 + 12
BLOCK 5 + !
```

9. General access to the (whole) memory

No good FORTH programmer would ever accept to be strongly restricted in his access to the memory. But if the " TO " concept is really accepted, then all the @ and ! operators should disappear. I suggest, instead, to add (its usage could be restricted) a generalized array called MEMORY (or any equivalent) with once more the double code associated with it. Then <address> MEMORY would either fetch from or store into the real address.

As an example, we would have

```
Ø TO 15317 MEMORY instead of
Ø 15317 ! or

DO I MEMORY S. LOOP instead of
DO I @ S. LOOP etc.
```

MEMORY is clearly equivalent to @ and ! depending on $VAR.

10. Generalization of address matching: virtual arrays

The previous points 8 and 9 suggest a further generalization that may solve another problem we have had since the beginning of FORTH: the use of arrays as parameters for a procedure. It was generally solved by putting the address of the first element on the stack, and doing explicit address arithmetic in the procedure (see the FPT of Jim Brault for example). This is certainly neither clean nor fast.

What I propose is the following: A virtual array (VARRAY , DVARRAY , FVARRAY , CVARRAY etc.) behaves like an array, but does not reserve space, except for a pointer to the real array. The link between the virtual and any portion of the memory is established by the word MATCH .

For example:

```
1ØØ ARRAY CUSTOMER
1ØØ ARRAY STAR
    VARRAY NUMBER
  , CUSTOMER MATCH NUMBER
```

associates the real array CUSTOMER with the virtual array NUMBER .

Then <i> NUMBER will be equivalent to <i> CUSTOMER.

Remember that, as previously,

<i> NUMBER pushes the value of the i^{th} STAR or CUSTOMER on the stack.

and <m> TO <i> NUMBER will store <m> into the i^{th} STAR or CUSTOMER .

At any time, CUSTOMER can be replaced by STAR , (and vice versa) by

```
' STAR MATCH NUMBER
```

In this way, MEMORY is defined by

```
          VARRAY   MEMORY
        ø MATCH    MEMORY
```

```
;S  PB
```

```
SCR # 150
  0 (  Example of the creation of TO  )
  1 HERE  0  ,  CONSTANT  ZVAR    !  ZVAR  SET  TO
  2 : VAR   CONSTANT  ;CODE  INB,  ZVAR  LDA,
  3        A0 IF,  B I) LDA,  PUSH,
  4             ELSE,  CLA,  ZVAR STA,  S) LDA,  B I) STA, POP,
  5             THEN,
  6 : DVAR   CONSTANT  ,  ;CODE  INB,  ZVAR LDA,
  7        A0 IF,  B LDA,  DSP,  A I) DLD,  S) STB,  PUSH,
  8             ELSE,  CLA,  ZVAR STA,  ..T STB,  S) DLD,
  9                    ..T I) DST,  POP.,
 10             THEN,
 11 : ARRAY   0  CONSTANT  DP  +!  ;CODE  INB,  S) ADB,  ZVAR LDA,
 12        A0 IF,  B I) LDA,  PUT,
 13             ELSE,  CLA,  ZVAR STA,
 14                    S1) LDA,  B I) STA,  POP.,
 15             THEN,

SCR # 151
  0 : 2ARRAY   0  CONSTANT  DUP  +  1+  DP  +!
  1       ;CODE  INB,  S) ADB,  S) ADB,  ZVAR LDA,
  2         A0 IF,  B I) DLD,  S) STB,  PUSH,
  3              ELSE,  CLA,  ZVAR STA,  ..T STB,
  4                     ISP,  S) DLD,  ..T I) DST,  POP.,
  5              THEN,
  6 : VARRAY  0 CONSTANT  ;CODE  INB,  B I) LDB,  S) ADB,  ZVAR LDA,
  7        A0 IF,  B I) LDA,  PUT,
  8             ELSE,  CLA,  ZVAR STA,  S1) LDA,  B I) STA,  POP,
  9             THEN,
 10 : DVARRAY  0 CONSTANT ;CODE INB, B I) LDB, S) ADB, S) ADB,
 11        ZVAR LDA,  A0 IF, B I) DLD,  S) STB,  PUSH,
 12             ELSE,  CLA,  ZVAR STA,  ..T STB,
 13                    ISP,  S) DLD,  ..T I) DST, POP.,
 14             THEN,
 15 FORTH   IMP  "  : MATCH  "  !  ;    IMP  '
```

Pauls' example of the use of TO will be presented in the next issue of Forth Dimensions. It utilizes Knuths' example for the calculation of the dates of Easter.

FORTH IMPLEMENTATION PROJECT

In June of 1978, the Forth Interest Group held its' first public meeting. With only mimimal publicity, we had in excess of 40 people attend. We had intended to offer educational assistance in using Forth. However, we found everyone was enthusiastic to learn Forth, but only five had access to running systems.

FIG then surveyed for vendor availability of the language. We found there were numerous mini-computer versions at educational and research institutions, all directly decended from Mr. Moore's word at NRAO. Of course, Forth, Inc offers numerous commercial systems.

None of these systems were available for personal computing. It appeared unlikely that this need would be met in the forseeable future. Our conclusion was that a suitable model should be created, and transported to individual micro-computers. Thus was born the Forth Implementation Team (FIT).

This team was proposed as a three tier structure. The first tier had several experienced Forth systems programmers, who would provide the model and guide the implementation effort. The next tier was the most critical. It was composed of systems level programmers, not necessarily having a background in Forth. They were to transport the common language model to their own computers by generating an assembly language listing that followed the model. Their results would be passed to the distributors that form the third tier. These distributors would customize for specific personal computer brands. Finally, the users could have access to both source and object code for maintenance.

Space doesn't permit inclusion of the FIT project, itself. The project was detailed as one of the six Forth conference papers at the Fourth West Coast Computer Faire, May 1979 in San Francisco.' [1] The result is that FIG now offers the Installation Manual with glossary and Forth model ($10.00) and assembly language listings for numerous computers

($10.00 @) Included are: 8080, PDP-11, PACE, 9900, 6800, and soon 6502, and Z-80.

Note that FIG offers these listings which still have to be edited into machine readable form, customized, and assembled for specific installations. We hope that local teams share the effort and then distribute for others.

Reports of installations are beginning to come in from the USA and Europe. We sincerely hope this work will give a benchmark of quality and uniformity that will raise the expectation of all users.

FIG would like to thank the following members of the Implementation Team who have devoted a major part of nine months' spare time to this effort.

Dave Boulton	Instructor
John Cassady	8080
Gary Feierbach	Comp. Aut.
Bernard Greening	Z-80
Kim Harris	Librarian
John James	PDP-11
Dave Kilbridge	PACE
Dave Lion	6800
Mike O'Malley	9900
Bill Ragsdale	Instructor
Bob Smith	6800
LaFarr Stuart	6800

[1] Ragsdale, William F.
"Forth Implementation, A Team Approach"
The Best of the Computer Faires, Vol IV
from: Computer Faire ($14.78, USA)
333 Swett Road, Woodside, CA 94062

FORTH INTERNATIONAL STANDARDS TEAM

For several years the Forth Users Group (Europe) has sponsored a team working toward a standards publication for Forth. The 1977 meeting (Utrecht) produced a working document FORTH-77. Attendees included European educational institutions and Forth, Inc.

In October, 1978, an expanded group met at Catalina Island (Calif.). Attendees included Forth Users Group (Neiuwenhuizen, Bartholdi), Forth, Inc. (Moore, Rather, Sanderson), FIG (James, Boulton, Ragsdale, Harris), Kitt Peak (Miedaner, Goad, Scott), U of Rochester (Forsley), SLAC (Stoddard), and Safeguard Ind. (Vurpillat).

The document resulting from this four day meeting has been released as FORTH-78. This document is becoming a good reference guide in evaluating the consistency and completeness of particular Forth systems. It is available from FIST to participating sponsors. (See below.)

A major benefit of the team meeting was the development of close communications between major users. For example, FIG is learning from the multi-tasking of U of R, Kitt Peak and Utrecht are running fig-FORTH, and we have adopted the security package pioneered in Europe. None of these events would have been likely without the contacts begun at Catalina.

The Team has announced the next Standards Meeting for October 14 thru 18, 1979, again at Catalina. The team agreed on an orginizational budget of $1000.00, to be met by $30. contributions by sponsors (individuals and companies).

These funds will be used solely to defray organizing costs of the annual meeting and distribution of the working documents to participants. Those considering participating should become Team Sponsors by remiting as given below. Sponsors will receive the just released FORTH-78, and all Team mailings.

Please remit to FIST, %Carolyn Rosenberg, Forth, Inc. 815 Manhattan Ave., Manhattan Beach, CA 90266.

PASCAL COMPUTING SERVICES, INC.

1979 February 22

Editor
FORTH DIMENSIONS
P.O. Box 1105
San Carlos, CA 94070

Dear Sir:

I was somewhat disconcerted when I read the article by Mr. David J. Sirag, "DTC Versus ITC for FORTH on the PDP-11", FORTH Dimensions, Volume I, No. 3. The author has, I believe, misunderstood the intent of the article by Mr. Dewar.

In Mr. Dewar's article, the definitions of direct threaded code (DTC) and indirect threaded code (ITC) are

> "DTC involves the generation of code (my emphasis) consisting of a linear list of addresses of routines to be executed."
>
> "ITC..." (involves the generation of code consisting) "...of a linear list of addresses of words which contain addresses of routines to be executed."

As applied to the FORTH type of heirarchal structure (Heirarchal indirect threaded code?), I would extend Mr. Dewar's definition to be

> "ITC involves the generation of code consisting of a linear list of addresses of words which contain addresses of routines to be executed. These routines may themselves be ITC structures."

However, Mr. Sirag based his conclusions on the following loose definition:

> "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."

Obviously the two men are not referring to the same things. Mr. Dewar is referring to the list of addresses which define the FORTH word, while Mr. Sirag is referring to the implmentation of the FORTH interpreter. If indeed Mr. Sirag's statement were true (which it is not) that their "analysis contradicts the findings of Dewar", then they should have implemented a DTC language rather than the ITC language of FORTH! Indeed, a careful examination of what is actually occuring in LABFORTH reveals that their techniques are logically identical to Dewar's ITC. They have simply, through clever programming, taken advantage of a particular instruction set and architecture. It is beyond the scope of this letter to prove this equivalence, or to support the FIG desire to have a common implementation structure for all versions of FIG FORTH.

Please note that I am <u>not</u> quibling over semantics with Mr. Sirag. All definitions are arbituary. (However, the value of a definition lies in its consistency, precision, and useability. I find Mr. Sirag's definition of DTC and ITC to be inconsistant with the environment in which he operates, FORTH, and thus quite useless.) My intent is two fold: (1) I am a self appointed defender of the excellent work of Mr. Dewar, and (2) I want to correct any misconceptions concerning this issue for readers of this newsletter who did not have access to Dewar's (better) definition of DTC and ITC.

Sincerely,

Jon F. Spencer

Jon F. Spencer
President
Pascal Computing Services, Inc.
14011 Ventura Blvd., Suite 201E
Sherman Oaks, CA 91403

14011 VENTURA BOULEVARD • SUITE 201E • SHERMAN OAKS, CALIFORNIA 91403 • (213) 995-4238
636 BROADWAY • SAN DIEGO, CALIFORNIA 92101 • (714) 231-4852

poly-FORTH BY FORTH, INC

Because of its speed and economy, FORTH Inc.'s FORTH language has been favored by many mini and microcomputer designers. Now comes polyFORTH, which combines the best features incorporated in the mini and micro versions.

PolyFORTH is a multilevel language with the essential functions (e.g., basic arithmetic and logic operators) in a 512-byte nucleus, and user-defined "words" (comparable to macros) in the outermost layer. What's more, the standard package fits in 4 kbytes of PROM, with an additional 2 k for the assembler and text editor (which aren't needed to run a program once it's developed).

The new operating system goes beyond previous FORTH versions by being able to handle multitasking and many terminals (limited only by the hardware), and by including a buffer handler that supports RAM or mass storage. Other improvements over previous versions include faster dictionary search, all 16-bit arithmetic and a simpler target compiler.

The target compiler can be used to develop a program of a micro-computer development system. The compiler code can either be executed directly, or be compressed and burned into ROM. Scientific routines, a data-base management system and applications software are available options.

The 8080 and 9900 are polyFORTH's first targets, but versions for the 8086, LSI-11, Series-1 and Honeywell Level 6 are scheduled by the Manhattan Beach, CA company for release later this year.

Contact Steven Hicks at FORTH, Inc., 815 Manhattan Avenue, Manhattan Beach, CA 90266, (213) 372-8493.

January 23, 1979

Dear FIG:

Having just received your issue No. 3, it seems to me that the "DTC" method used by Sirag is still "indirect threaded code" in the terminology used by Dewar, and should be distinguished from other implementation methods as being "executed" rather than "interpreted." The use of actual machine code in place of the code address is the "executed" aspect, but only in the case of the Low Level Definition (Code) does the use of machine code reduce the level of indirection in the threading to that of direct threaded code. In the Storage Definition in Sirag's diagram, there is still a subroutine call in the dictionary entry containing data (constant, variable type entries). Direct threaded code would require this subroutine call to be moved from the individual data-Word to the code string referencing that word. Whether that subroutine call is executed by an actual machine language JUMP SUBROUTINE instruction or by an interpreter routine is another matter. Now, in the case of the Code type dictionary entry, use of executed machine code tends to also remove a level of indirection because only one jump is needed, there being only one address, that of the code routine, involved; this coincidence unfortunately confuses the two consequences, even though they are separate.

The concept of Threaded Code seems inherently fuzzy, because any high-level instructions compiled by a translator into a series of subroutine calls has the same form as threaded code, so it looks like almost any compiler is going to use a certain amount of threaded code. Some operations, however, like adding two numbers, take so little machine code that many compilers would expand them completely in-line instead of threading them in a subroutine, and it is at this low level of primitives that the concept has meaning. Even when a subroutine call saves nothing compared to a full in-line expansion, going to two subroutine calls, i.e. indirect threaded code, many save space by reducing the amount of code to identify the types of the operands being processed, and in some cases it may also save time. On a very large computer perhaps all the variable types involved are an inherent part of the machine instruction set, and the memory may be tagged to identify type there as well, and both the questions of executed/interpreted and indirect/direct would not be relevant. If one had a machine which executed FORTH primitives as its machine language, you would still have indirect threaded code, but completely executed instead of interpreted. The PDP-11 seems to fall in between.

None of the above considerations, however, contradict Mr. Sirag's main conclusion that how FORTH is implemented should not be part of its definition.

One final note: in the DTC, ITC comparison for storage definitions, DTC was shown having a larger overhead than ITC even though the VAR routine appears shorter in the DTC case. Even though the space overhead is greater, I wonder if he has overstated the DTC overhead time. He seems to show a single jump-subroutine taking longer than an interpreted jump.

Sincerely yours,

George B. Lyons
280 Henderson Street
Jersey City, New Jersey 07302

November, 1978

Dear FIG:

I was very excited to find something from you in my mail today, but then I was disappointed to discover that it was a copy of the journal article which introduced me to FORTH and your existence, which I already have.

On October second, I sent you a check and asked for everything else offered on the subscription form (FORTH DIMENSIONS, Volume 1, number 1, p. 22.), i.e., newsletter sub., glossary, and FORTH-65. And I've been anxiously awaiting the receipt of any or all of these.

Of course I realize you're all volunteers, and I'm not angry...but I really would like to get that stuff. I am, like many others, I imagine, anxious to get a version of FORTH up on my system. I've managed to dig up most of the references listed in the article (still waiting for DECUS) except for the last one: is it in print?

I also have the documentation for the Digital Group's CONERS and Programma's 6502FORTH (for the Apple, which I don't have). Both of these programs are outer interpreters written in assembly language, and contain no inner interpreter. Interesting, and they look like FORTH, but that's not really what I want to do...

Rather than invest another 30 cents in postage, I'm enclosing a check for $2.00 for the reprint--I know these things aren't free, but pleaes send the other items soon, okay? Thanks.

Sincerely, Dan del Vecchio
Ann Arbor, MI

Editor --

Apologies to Mr. del Vecchio. We mishandled his entire request, and yet he encloses an extra $2.00! Our mail processing is now current. We will complete the full six issues of FORTH DIMENSIONS.

```
SCR #  3

  0                GLOSSARY DOCUMENTATION - D.W.BORDEN
  1     AFTER READING W.F.RAGSDALE'S ARTICLE ON THE "HELP" COMMAND
  2 IN FORTH DIMENSIONS NO.2 AND SOME PROPAGANDA FROM FORTH INC., I
  3 WROTE A SHORT PROGRAM WHICH PRINTED OUT EACH FORTH WORD AS IT
  4 IS DEFINED, THE ADDRESS OF THE LENGTH BYTE AND THE ADDRESS OF
  5 THE PARM BYTE. AFTER EACH ENTRY, I PRINTED ELLIPSES TO ALLOW
  6 A HANDWRITTEN ENTRY OF WHAT EACH COMMAND IS SUPPOSEDLY DOING.
  7 THE ADDRESS OF THE PARM BYTE IS USEFUL SINCE THAT ADDRESS
  8 APPEARS IN HIGH LEVEL COLON DEFINITIONS "OBJECT" CODE. THUS, IF
  9 YOU HAVE NOT WRITTEN YOUR OWN FORTH SYSTEM, YOU CAN HAND
 10 DISASSEMBLE ( DISFORTH MIGHT BE MORE APPROPRIATE ) EACH COMMAND
 11 AND SEE WHAT IT IS DOING.
 12    OF COURSE A DISFORTH PROGRAM COULD BE WRITTEN AND I HAVE DONE
 13 SO, BUT I AM NOT HAPPY WITH ITS OUTPUT FORMAT YET. I ALSO HAVE
 14 WRITTEN A TRACE WHICH PUTS A TRAP IN "NEXT" AND LISTS EACH FORTH
 15 COMMAND EXECUTED. VERY USEFUL FOR DEBUGGING.     ;S  01/31/79
SCR #  2

  0 ( GLOSSARY OF FORTH WORDS WITH HEAD AND PARM ADDRESSES )
  1 0 VARIABLE CMD  ( TEMPORARY VARIABLE TO HOLD COMMAND )
  2 : TOPOFPAGE CR CR [ CMD    HEAD  PARM ] CR ;
  3 : UNDERLINE [ ..................................................... ] ;
  4 : GLOSSARY TOPOFPAGE CURRENT @ @ CMD ! ( GET TOP CMD ADDRESS )
  5 BEGIN
  6 CMD @ IF CMD @ C@ DUP 80 AND - ( GET COMMAND LENGTH )
  7 DECIMAL . HEX 4 1 DO ( PRINT COMMAND LENGTH )
  8 CMD @ I + ( INDEX FETCHED IS 1-2-3 )
  9 C@ ECHO ( PRINT COMMAND LETTER )
 10 LOOP SPACE
 11 CMD @ .H SPACE ( PRINT COMMAND HEAD ADDRESS )
 12 CMD @ 6 + .H UNDERLINE CR CR ( PRINT CMD PARM ADDRESS )
 13 CMD @ 4 + @ CMD ! ( PICK UP LINK WITH NEXT CMD ADDRESS )
 14 ELSE QUIT THEN
 15 AGAIN ;              ;S       1/30/79       DWB

 GLOSSARY

 CMD    HEAD  PARM
 8 GLO  1C2A  1C30.....................................

 9 UND  1BEE  1BF4.....................................

 9 TOP  1BC9  1BCF.....................................

 3 CMD  1BBD  1BC3.....................................

 9 ?TE  1BA1  1BA7.....................................
```

example	fig-FORTH
[	."
]	"
ECHO	EMIT
.H	H. (hex output)

Dear FIG:

I have been programming for sixteen years, but I only discovered FORTH about two weeks ago! It was a clear case of love at first sight, but I've encountered a really sticky problem already. I have seven different programs, each runs on my Apple II computer, and each claims the name "FORTH". Aside from that the only thing they have in common are the three verbs, ":", ";", and "@". Beyond that, everything is different.

My question is: Is there such a thing as a "standard" FORTH, and if so, where do I find out more about it? The only documentation I have rounded up so far is one borrowed copy of FORTH DIMENSIONS, and two pitifully incomplete "user's guides" supplied with two of the versions FORTH I've bought. HELP!

Sincerely,

Gary J. Shannon
14115 Hubbard Street
Sylmar, CA 91342

Editor --

Mr. Shannon addresses a major problem! FORTH uniformity is a major problem. Current standards (FORTH-77 and FORTH-78) exist but cover only areas of mutual user concensus. There are remaining areas where definition and/or refinement are needed. The FIG Installation MANUAL has a model of the language offered for public comment toward uniformity. We also pin-point areas of deviation from FORTH-78 in our publications.

Dear FIG:

I have an 8k PET and after reading Dr. Dobb's number 28 I called Programma Consultants to purchase FORTH. Two weeks later I received it!

I have been working with PET FORTH now for a couple of weeks and am still fascinated with FORTH although I have some problems with Programma's implementation.

Please sign me up.

Regards,

Chris Torkildson
St. Paul, MN

December, 1978

Dear FIG:

The FORTH-65 implementation listing you sent is great!!! I'm beginning to understand the power and beauty of this language/system, with the aid of it, a Caltech (Space Radiation Lab) FORTH manual, and James' article in Dr. Dobbs...but there are still some items I can't figure out. (One stupid question: What is PROTECT, used in screens 36 and 45?)

Also, have you reviewed Programma International's PET FORTH yet? They told me that a new version (1.1) would be out in Nov./Dec., but I'm waiting before buying...would like your opinions/recommendations.

Best,

Mark Zimmerman
Caltech 130-33
Pasadena, CA 91125

Editor--

Programma International's version is not recommended by FIG. It has a non-standard header (no word length indication) and is pure machine code. The inner interpreter NEXT is missing as are the critical definitions ;CODE, <BUILDS, DOES>, and BLOCK. PROTECT traps execution of compiling words outside of colon-definitions. FIG now uses ?COMP for this purpose.

April, 1979

Dear FIG:

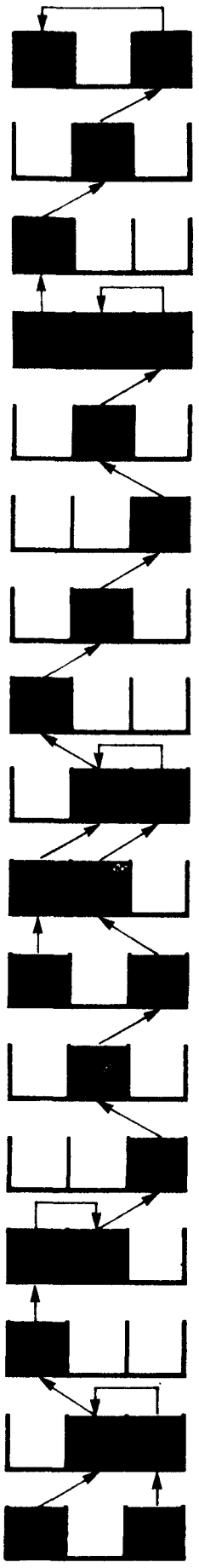
I now have Programma's 6800 FORTH and I probably shouldn't complain, however it does have some differences with the DECUS FORTH (CALTECH FORTH). I think my major complaint is with the coding; any conditional branching is almost impossible without a previously assembled listing to obtain address displacements. My MIKADOS/OS/assembler/disassembler) is interactive, single-pass, and fast. I haven't figured out yet how to do any editing other than backspace and line deletion, other than redo the program. Also, I have the Felix USP and recursive programs just come naturally; same with textual programming - somewhat difficult with FORTH. Can recursive work be done in FORTH? I don't know as I can't find out enough from any manual (the only one with Programma's is "interim") to answer my questions.

My "down" outlook may surprise other Figgers, however I am (obviously) not a computer scientist but feel that after the Dobb's puffery by John James and a couple of letters to the editor, the simplicity and all-encompassment have been vastly overrated.

Sincerely,

Neal Chapion
Space #27
602 Copper Basin Road
Prescott, Az 86301

Editor--
The above letters illustrate the negative comment we receive about Programma Internationals' FORTH.



FORTH DIMENSIONS

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

Volume 1
Number 5
Price $2.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume 1 No. 5 Jan/Feb 1980

Publisher Roy C. Martens

Editorial Review Board

Bill Ragsdale
Dave Boulton
Kim Harris
John James
George Maverick

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL MATERIAL PUBLISHED BY THE FORTH INTEREST GROUP IS IN THE PUBLIC DOMAIN. Information in FORTH DIMENSIONS may be reproduced with credit given to the author and the Forth Interest Group.

Subscription to FORTH DIMENSIONS is free with membership in the Forth Interest Group at $5.00 per year ($9.00 overseas). For membership, change of address and/or to submit material, the address is:

Forth Interest Group
P.O. Box 1105
San Carlos, CA 94070

HISTORICAL PERSPECTIVE

FORTH was created by Mr. Charles H. Moore in 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of dissatisfaction with available programming tools, especially 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.

The Forth Interest Group is centered in Northern California, although our membership of 950 is world-wide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

PUBLISHER'S COLUMN

Forth Interest Group has come of age. FIG now has a publisher for FORTH DIMENSIONS. Roy Martens will handle all facets of putting together and getting out future issues of FORTH DIMENSIONS. He comes to FIG with a solid background dating back to Hughes Aircraft Company in the 50's to Singer Business Machines and American Micro Systems in the 70's. He has publication experience gained from Hayden Publishing Company (Electronic Design, Computer Decisions, etc.), CBS and EW Communications. Welcome aboard, Roy!

S. Figgie

Thanks. I look forward to working with FIG and hope that we can make FORTH DIMENSIONS a useful and timely tool for all members. Please send in your letters, cases, suggestions. Your input will make FORTH DIMENSIONS successful.

Roy C. Martens

$$$$ CASE STATEMENT $$$$ CONTEST

FIG is sponsoring a contest for the best CASE statement for FORTH.

Prize.....$100 ($50 from FIG and $50 from FORTH, Inc.)

Furthermore, entries will be considered as experimental proposals for possible inclusion in the future FORTH Standard.

Contest Rules.....

Submit a CASE Statement, as specified below, to

FIG CASE CONTEST
P.O. Box 1105
San Carlos, CA 94070

Postmarked on or before March 31, 1980.

All entries will be judged by selected, non-entering members of Forth Interest Group. All entries will become public domain and may be published by FIG.

Judging Criteria.....

A. Conformity to rules
B. Generality of statement
C. Simplicity of statement
D. Self-identifying function
E. "FORTH-like" style

Entry Requirements.....

Your entry should contain descriptions of a collection of FORTH words which allow the selection of one of several actions based on a selection criteria. The actions may be single words or groups of words. The selection criteria may be simple or complicated. A variety of situations should be accommodated. Included in your entry should be....

A. An overview of the statement

B. Source definitions in fig-FORTH words for the needed compiler and support words

C. An "English" explanation of how these words work

D. Glossary entries for each word

E. Examples of the use of this statement

F. A discussion on the statement, including advantages and disadvantages, limitations, applications, etc.

Contest Purpose.....

The selection of one of several procedures based on some criteria is a useful and common control structure. Standard FORTH provides two mechanisms for this structure: nested IF structures and execution vectors. It is desirable to have a standard structure to handle a variety of situations.

However these do not appropriately satisfy all situations in terms of source convenience and execution-time efficiency. A simple situation where the selection criteria is a single integer index with a limited, continuous range is adequately met by FORTRAN's computer GOTO or FORTH's execution vectors

At the other end of the complexity scale, if the selection criteria was an index with a non-contiguous range or a series of expressions, the simple statements would require manipulations at the source level and would appear less clear. Because of FORTH's hierarchial modularity, the range of complex situations can be met with a range of structures. The execution-time overhead need not be greater than what the situation requires. The purpose of this contest is to produce a "kit" of compiler words which will allow the optimum specification of case control by combining minimum execution-time overhead with a uniform source language.

;S

"TO" SOLUTION CONTINUED..... "EASTER"

As promised in the last issue, here is the example for the calculation of the dates of Easter from Paul Bartholdi, Observatoire De Geneve, Switzerland.

SCR #9

```
 0 (Dates of Easter, Clavius Algorithm   )
 1 (This algorithm and variable names are)
 2 (from "Fundamental Algorithms" by     )
 3 (D. Knuth.  The method was originated )
 4 (by the sixteenth century astronomer  )
 5 (Clavius.                             )
 6
 7
 8 0 VARIABLE %VAR     ( if one, store   )
 9 : TO   1 %VAR !  ;  ( set to store    )
10 : FROM  0  %VAR ! ; ( set to fetch    )
13 : ((              (preserve %VAR flag )
14      R>  %VAR  @  >R   >R FROM  ;
15 : ))              ( restore %VAR flag )
14       R>   R>   %VAR  !  >R ;
15
16  —
```

SCR #10

```
 0 ( Simplified TO          DAB-79OCT29 )
 1
 2 : VARIABLE  ( defined to observe TO )
 3  <BUILDS   0  , DOES>  %VAR   @
 4    IF ( set, so store )  !  FROM
 5     ELSE ( clear, so fetch ) @ THEN ;
 6
 7
 8 VARIABLE C  VARIABLE D  VARIABLE E
 9 VARIABLE G  VARIABLE N  VARIABLE K
10 VARIABLE X  VARIABLE Z  VARIABLE YEAR
11
12
13  —>
14 Note that this demonstration does not
15 include the newer  +TO ,  AT  etc.
16
```

SCR #11

```
 0 ( Dates of Easter          DAB-79OCT29 )
 1 : (EASTER)        ( year —— day  day )
 2   (calculate date relative to March 1)
 3   DUP  DUP  TO  YEAR  19  MOD 1+ to  G
 4    100   /   1+   DUP   DUP   TO   C
 5   3   *  4  /  12  -  TO  X
 6    8   *   5   +   25   /   5   -   TO   Z
 7   YEAR  5  *  4  /  X  -  10  -  TO  D
 8   G  11  *  20  +  Z  +  X  -  30  MOD
 9   DUP  0<  IF  30  +  THEN
10   DUP  DUP  TO  E
11   25  =  G  1  >  AND  SWAP  24  =  OR
12   IF  E  1+  TO  E  THEN
13   44  E  -  DUP  TO  N
14    21  <  IF  N  30  +  TO  N  THEN
15   N  DUP  7  +  SWAP  D  +
16   7  MOD  -  DUP  DUP  TO  N  ;  —>
```

SCR #12

```
 0 ( Dates of Easter          DAB-79OCT29 )
 1 : EASTER  ( year ——print one Easter )
 2         (EASTER)  31  >
 3    IF   31   -   5   .R   ." APRIL  "
 4       ELSE   5   .R   ." MARCH  "  THEN
 5       YEAR   5   .R   ;
 6
 7 : EASTERS  (Begin year, end year —— )
 8  ( print Easter for a range of years.)
 9  PAGE 32 SPACES ." DATES OF EASTER" CR
10  1+  0  SWAP  ROT
11  DO  I  EASTER  1+
12     DUP   4   MOD   0=   IF   CR   THEN
13     DUP   240   MOD   0=   IF PAGE THEN
14  LOOP  CR  PAGE  DROP  ;
15
16
```

EXAMPLE

10	April	1955	1	April	1956	21	April	1957	6	April	1958
29	March	1959	17	April	1960	2	April	1961	22	April	1962
14	April	1963	29	March	1964	18	April	1965	10	April	1966
26	March	1967	14	April	1968	6	April	1969	29	March	1970

A MODEST PROPOSAL FOR DICTIONARY HEADERS

Robert L. Smith
Palo Alto, CA

The new fig-FORTH model has improved the utility of FORTH by allowing dictionary names to be of any length, up to the current maximum of 31 characters. Most previous implementations stored only the first three characters of the name, along with a count field. Confusion could easily arise, say between the words "LOOK" and "LOOP". The maximum word length stored in the fig-FORTH model can be changed dynamically by changing the value of WIDTH. If one wishes to return to the former restriction of 3 character names, the new model will still be an improvement because 1 and 2 character names will require less dictionary space.

One advantage of the old dictionary format has been lost, namely a fixed relationship between the link field and the CFA (Control Field Address) or the parameter field. The following proposal will restore the fixed relationship, while at the same time keep the new fig-FORTH advantages. In addition, the maximum allowable word size is increased to 63.

A model of the proposed dictionary entry is shown in Figure 1. The name of the word is stored in reverse order to simplify the dictionary searching. From the link field one can step backwards to obtain the name, or forwards to obtain the definition of the word. The leading bit of the machine representation of each character of the name is set to zero, except for the terminal character which has the leading bit set to one. Thus the effective end of the name can readily be determined. The name field is reversed to simplify the dictionary search procedure, but it is an implementation decision. Obviously one could put the characters in the forward direction, but the dictionary search would take longer. The suggested structure increases the maximum allowable word length to 63 (or possibly 64).

It would be possible to use the leading bit of the first character for the smudge bit, but it would somewhat complicate the dictionary search procedure. One character names would have to be a special case with no terminal bit specified, since that location would be designated for smudging purposes.

This proposal eliminates the need for a number of special words in the fig-FORTH model (TRAVERSE, PFA, NFA, CFA). The only obvious disadvantage is that the routine for printing the dictionary names would need to take characters in the opposite direction from other text. The advantages appear to outweigh the disadvantage.

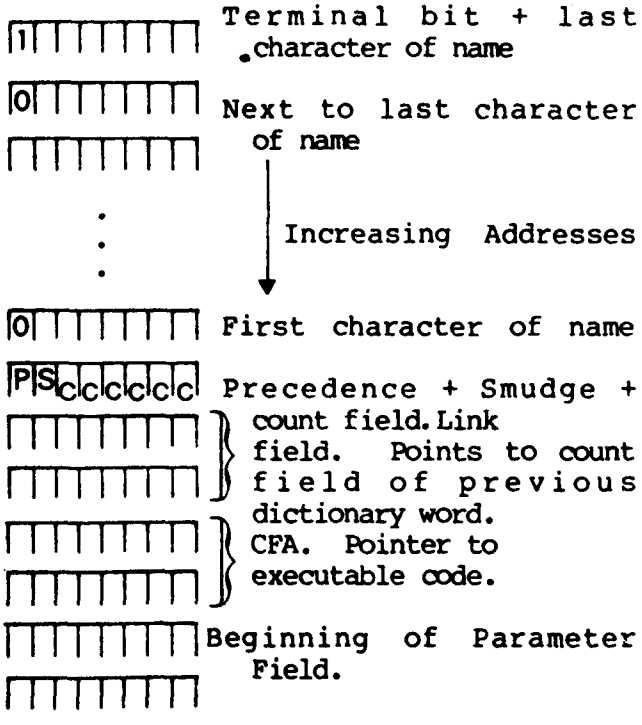


Figure 1. A proposed Dictionary Structure.

;s

FORTH-85 "CASE" STATEMENT

Richard B. Main
Zendex Corp.
Dublin, CA

NEPTUNE UES has recently extended its FORTH-85 with a "CASE" statement. The CASE statement allows an n-way branch based on a condition. Its use is very similar to the PASCAL CASE statement. the IF..(this)...ELSE... (that)... THEN structure is a two-way branch while the FORTH-85 DO-CASE END-CASES allows 65,000 different cases randomly arranged. (n+1 CASE may precede n CASE.) Each case can test on a 16-bit quantity. The CASE structure must begin with "DO-CASE" then "CASE...END-CASE" and finish with "END-CASES".

The test "CASE" structure screen gives an example of using CASE within FORTH-85. An unknown variable is passed to "MONITOR" for an n-way branch based on a match between the variable and one of the cases. "DO-CASE" places the variable passed on the stack into a location in memory. "41 CASE" will fetch the variable and compare it to 41. If it is 41, code between "41 CASE" and "END-CASE" will be executed (in this instance "ASSIGN" will be printed) and then a direct jump to "END-CASES". If the variable is not 41, then "41 CASE" will cause a jump to the next case, and so on.

The "n" for "CASE" need not be in-line code, nor absolute, nor known at compile time. During run-time "n" may be computed, fetched or otherwise placed on the stack just prior to executing case. The possibilities that exist for this, combined with 16-bit CASE testing and 65,000 possible cases AND (!) no restrictions on the amount or type of code between CASE... END-CASE, are immense. Some uses that occur to me immediately are monitor program executives, machine code disassemblers, text interpreters, and disk I/O drivers.

HOW IT WORKS (during compile)

The screen showing DO-CASE through END-CASES must be loaded into your system before they can be used. This screen will actually extend your FORTH compiler beyong having IF...ELSE... THEN, BEGIN...IF...ELSE...WHILE, DO... LOOP, AND BEGIN...END.

Line 1 extends the assembler to include JNC for use by the following code statements.

Line 2 contains code to set up the run-time variable "VCASE" for use by "DO-CASE" and "CASE".

Lines 3-9 code statements define the run-time behavior of "DO-CASE, CASE and END-CASE". While lines 11-15 define the compile-time behavior of the compiling words: "DO-CASE, CASE, END-CASE, and END-CASE".

During compile time "Do-CASE" assembles the code "DO-CASE", places "HERE" and Ø on the compile-time stack, and assembles (temporarily) a 16-bit Ø. "CASE" then compiles code "CASE" swaps the compile-time stack places "HERE" on the stack to locate "CASE" for "END-CASE" and temporarily assembles an 8-bit zero. "END-CASE" now has passed to it the location of "CASE" and "DO-CASE". "END-CASE" will pass "DO-CASE" location on to "END-CASES". "END-CASE" assembles code "END-CASE", places "HERE" on the compile-time stack for "END-CASES", computes "END-CASE" minus "CASE" and loads the result (assembles difference) at "CASE" +2. "END-CASE" further assembles a 16-bit zero temporarily for "END-CASES". "END-CASES" has passed to it, on the compile-time stack, all the locations of "END-CASE" and the beginning "DO-CASE". Since the number of cases is variable "DO-CASE" had put a zero on the compile-time stack to mark the bottom location the "BEGIN HERE SWAP! - DUP Ø = END" takes every "END-CASES" location and assembles "END-CASES" location there for direct forward jumps from "END-CASE" to "END-CASES".

The effect of all the compiling machination was to place 16-bit and 8-bit code for use by the address interpreter during execution. Reviewing the intrepreter during execution. Reviewing the interpreter: compiled colon definitions are simply strings of addresses of routines to execute. Therefore, the address string compiled for: "DO-CASE...CASE...END-CASE... END-CASES" is:

Fig 1

AB....CD....EF....(G)....
where:

A = 16-bit address of code "DO-CASE"
B = 16-bit address (G)
C = 16-bit address of code "CASE"
D = 8-bit value of distance F+1
E = 16-bit address of code "END-CASE"
F = 16-bit address of (G) and
(G) is the location to continue execution after case.
I = interpreter pointer

HOW IT WORKS (during execution)

The following will refer to A through (G) in Figure 1.

Before executing (address interpreting): A (DO-CASE) the case number to execute is placed on the run-time stack. Code DO-CASE is executed when A is encountered and it will pop the stack and store it at the memory location "VCASE". At this point there is no jump to make based on condition so I is incremented past B. B exists to provide a backward stub location for the compiling of the structure. Code between B and C is executed, and by the time C is encountered the case condition to test for is on the stack. C is the code "CASE" which will pop the stack and do a 16-bit compare to "VCASE". If there is no match code, "CASE" increments I (interp. pointer) by D (8-bit). Control would then pass to F+1. If there is a match, I is incremented by one and points to D+1. Code till E is executed. E will cause code "END-CASE" to execute and it will load I with F. Now I points to (G).

The code described here may be used for your personal use and experimentation only. Commercial users should write to the author.

EXAMPLE

```
 0 (  DO-CASE  CASE  END-CASE  END-CASES  )
 1 ASSEMBLER DEFINITIONS HEX : JNC D? END ;   FORTH DEFINITIONS
 2 0 VARIABLE VCASE
 3 CODE DO-CASE  H POP  ' VCASE SHLD  I INX  I INX  NEXT JMP
 4 CODE CASE  H POP  ' VCASE LHLD  L A MOV  M 1+ CMP
 5   0= NOT IF  I LDAX  I 1+ ADD  A I 1+ MOV  NEXT JNC
 6              I INR  NEXT JMP  THEN  H A MOV  M CMP
 7   0= NOT IF  I LDAX  I 1+ ADD  A I 1+ MOV  NEXT JNC
 8              I INR  NEXT JMP  THEN  I INX  NEXT JMP
 9 CODE END-CASE  I LDAX  A L MOV  I INX  I LDAX  A H MOV
10   H PUSH  I POP  NEXT JMP
11 : DO-CASE   \ DO-CASE  HERE  0  0 ,  ;  IMMEDIATE
12 : CASE   \ CASE  SWAP  HERE 0 C, !  IMMEDIATE
13 : END-CASE   \ END-CASE  HERE 0 , SWAP  HERE
14    OVER  - SWAP  C! ;  IMMEDIATE
15 : END-CASES   BEGIN HERE SWAP !  -DUP  0 = END ;  IMMEDIATE
```

```
 0 ( TEST * CASE * STRUCTURE )    BASE C@   HEX
 1
 2 : MONITOR   DO-CASE
 3    41 CASE  [ ASSIGN ]      END-CASE
 4    44 CASE  [ DISPLAY ]     END-CASE
 5    46 CASE  [ FILL ]        END-CASE
 6    47 CASE  [ GO ]          END-CASE
 7    49 CASE  [ INSERT ]      END-CASE
 8    53 CASE  [ SUBSTITUTE ]  END-CASE
 9             END-CASES  ;
10
11 : KEYBOARD  BEGIN  KEY 7F AND  DUP MONITOR 20 = END ;
12
13 BASE C!
14
15
```

Editors Note:

This article has been presented as a good example of the ease of addition of control structures to match application needs.

FORTH-85 is a UES software product derived from microforth (TM, FORTH, Inc.). Both these versions contain definition names at variance with fig-FORTH and FORTH-78. We present a reference table:

FORTH-85 microFORTH	fig-FORTH	FORTH-78
[	."	."
]	"	"
	COMPILE	None
I	IP	None
END	UNTIL or END	END
THEN	ENDIF or THEN	THEN

;S

ANOTHER GENERATION OF MISTAKES?

Roger L. Gulbranson
University of Illinois

Much has been said about how each generation of computers the large mainframes, the minis, and now the micros - has repeated the mistakes of the past generation. There have even been comments on upward-compatibility mistakes in going from one generation of microprocessor to its succeeding generation(s).[1] I would like to take this further by commenting on the latest generation of microprocessors, the 16-bit CPUs.

I was talking to an EE who tried to convince me that the yet-to-be released microprocessors are "so much better" than the existing ones because they have included all of the addressing modes in each instruction. Among other things, I am told, this reduces program size and makes the micro run faster, since its speed is directly related to the number of fetches it must do per instruction and the number of instructions used. As a concept, in toto, I can only reply, BUNK!

If one were to design a microprocessor stack computer, [2] it would be possible to incorporate an instruction set that has only one multi-addressing mode instruction, a "load effective address" instruction. Since this is perhaps a bit austere, it may be realistic to add appropriate load to stack, store from stack, conditional and unconditional branch, and subroutine call instructions to the group of "addressed" instructions. The remaining instruction set need not contain more than 128 (if even that many) zero-address instructions. This instruction set can be arranged so that all zero-address instructions are 8 bits long. This means that most of the time an instruction word will contain two instructions, decreasing the number of memory cycles per instruction. If, in addition, the stack is "cached" on chip, the number of memory cycles per instruction will drop considerably. And if this latter idea is properly extended to the instruction stream to create an "instruction stack," much like that on the CDC Cyber computer line or the Cray-1, the number of memory cycles can be reduced even further. This reduction of memory cycles should noticeably increase the speed of our hypothetical microprocessor.

Considering the impetus given to virtual stack machines by the Pascal P-code groups[3] and the Concurrent Pascal originators,[4] one wonders why these ideas have not been efficiently implemented in silicon. Must we wait for another generation?

[1] L. Armstrong, "16-bit Wave Gathering Speed," Electronics, Vol. 51, No. 4, Feb. 16, 1978, pp. 84-85.

[2] A good overview of stack machines can be found in the May 1977 issue of Computer.

[3] References to these groups can be found in Pascal News, a publication of the Pascal User's Group.

[4] Per Brinch Hansen, The Architecture of Concurrent Programs, Prentice-Hall Inc., Englewood Cliffs, N.J., 1977.

Reprinted from Computer, April, 1979
Copyright 1979, IEEE

;s

INSTALLATION REPORTS

The distribution of fig-FORTH began on May 11, 1979 at the Fourth West Coast Computer Faire. The first installation to be brought up by a user occurred while the Faire was still running! Bob Steinhaus of Lawrence Livermore Lab got the 8080 listing on Saturday at the Faire. His wife read the hex code to him and he typed it in. By Sunday morning, he was running!

The next to run was Dwight Elvey of Santa Cruz. He organized five programmers on five Intel developmental systems. Each edited in 1/5 of the source code. They then merged the files, assembled at the listing origin and caught final editing errors by a byte-by-byte comparison. They then updated the I-O and re-origined. They were running in four evenings work (of five people).

The next to run was Dave Carlton of Ceres, CA. Dave brought up fig-FORTH on his TRS-80. He did fill his edit buffer and then crash, which cost re-typing 25 pages! Dave demonstrated his system to FIG on June 25. He had just interfaced to his floppy-disc and had the sample editor running.

John Forsberg of Maracaibo, Venezuela should be up and running with his Prolog 8080 with 32K RAM and casette, floppy and disk capability that he put together himself.

Frank D. Dougherty of Belvidere, IL has a IMD05 Ver 2.05 with 15 languages running. He says his AXion EX-801 printer works like a charm.

Many installations of the PDP-11 are operating. There is a short cut in this case. John James (the implementor) is also acting as a distributor. He provides a source diskette that will assemble and run under RT-11 and RSX-11M. Between 10 and 20 installations should be up by now. John has gone to great care to create a "portable" version for the various CPU variations. fig-FORTH is known to be running on Heathkit and DEC LSI-11's up through the DPD-11/60. Installations are known in California, Arizona, New York, and the Netherlands.

;s

MEETING NOTICES

LONDON, ENGLAND

FORMAL, a meeting to gather input for the future implementation of FORTH, will be held January 8, 9, and 10, 1980 at Imperial College, London, England. Attending from FIG will be: Kim Harris, Bill Ragsdale, Jon Spencer, Larry Forsley and probably 2 or 3 more. Look for a report in the next issue of FORTH DIMENSIONS.

NORTHERN CALIFORNIA

FIG monthly meetings will continue to be held the fourth Saturday of each month at the Special Events Room of the Liberty House department store in Hayward. Informal lunch at 12 noon at the store restaurant, followed by the 1 pm meeting. Directions: Southland Shopping Center, Highway 17 at Winton Avenue, Hayward, CA, Third floor, rear of the Liberty House. Dates: 1/26/80, 2/23/80, 3/22/80, etc. All welcome.

Send us notices of any meetings that you know about.

;s

LETTERS

Despite your warnings regarding Programma International's implementation of FORTH for the PET, I bought it just to have something to work with and get a feel for FORTH (or at least a FORTH-like system), and I have been having a ball with it (despite the limitations of this version), after adding 16K of RAM - I thought the thing would run in 8K but found out differently. The first VOCABULARY written for the PET was a DEFORTH routine to disassemble the dictionary. After DEFORTHing the latter dictionary entries and saving them on tape, I was able to truncate the dictionary to wipe out 10 unneeded words. This saved about 3 pages of memory.

Edward B. Beach
Arlington, VA

Editor...
People keep trying!

You will find enclosed a set of source listings for the 8080 nucleus of MSL. (Editor's note: Incorporated in fig-FORTH 8080 Assembly Listing, Version 1.1 see New Products.)

All my code routines are re-entrant. Not only that, but without exception, they use no more bytes or time than Cassady's routines. In fact, in many cases, you will note that my routines use fewer bytes and run faster. In some cases, such as multiply and divide, the improvement is enormous! viz

U* UNSIGNED MULTIPLY

4950 cycles	81 bytes	CASSADY
994 cycles	47 bytes	VILLWOCK

U/ UNSIGNED DIVIDE

30,600 cycles	203 bytes	CASSADY
2,495 cycles	76 bytes	VILLWOCK

I was well into the design and coding of MSL when I stumbled upon FIG and its efforts. Your documents (particularly the installation manual and James' work on the 11) have been most useful and have saved me considerable time in putting the finishing touches on MSL. I'd like to return the favor, so I hope that my 8080 routines will be useful to you.

Many thanks again to FIG for your excellent efforts.

R. D. Villwock
Pasadena, CA

MORE FROM GEORGE

Following are some observations on PASCAL and FORTH implementation details made upon reading the recent article in Dr. Dobb's Journal.

With the emergence of a low cost yet elaborate compiler for PASCAL at USCD consideration might be given to PASCAL as an alternative to FORTH. PASCAL is implemented with a self-compiling, virtual-machine language system offering transportability similar to FORTH, and is supplied by USCD with full source code. From the standpoint of sophisticated extensions to a system such as high-speed arithmetic processing hardware, converting from floppy-disks to hard disks, adding memory beyond directly addressable space, however, PASCAL may be much more difficult to work with. The system involved very large programs including a compiler, an interpreter, a run time

executive, and a debugger--all of which must be consistently modified in making extensions. The USCD authors may well turn out to be the only users able to efficiently make expansions themselves.

An important advantage seems present in PASCAL, however, which is dynamic storage allocation--the nested globals and locals environment, and FORTH does not seem to offer this. Several methods for accomplishing this in FORTH might therefore be noted. First, existing FORTH system words can be used to create a class of variables and constants whose code-address points to a new defining-word which returns, not the address or data in the variable or constant itself, but in a dynamically reserved area on the stack. The parameter field of the variable holds an address offset generated at compile time, relative to an environment pointer implemented as another variable whose parameter field is filled with the stack pointer value upon entry to a procedure, before shifting the SP to reserve the data area. The code addressed by the variable combines the offset with the environment pointer and returns either this address, or the data at that address on the stack.

The process of retrieving dynamically stored data would be slowest when implemented with regular FORTH language, so a second method is to provide machine language for the basic mechanisms involved just as FORTH does for its standard data handling. Assumed in both cases is a provision for compiling words (e.g. SVARIABLE--define a "stack" variable) which creates the related dictionary entries with the same ease as the regular VARIABLE and CONSTANT words do.

An interesting variant would be pointing the code-address of a variable at a modifiable jump-vector. Initially the vector points to code which reserves space for the variable on the stack, and is then switched to point the code retrieving the data. Thus storage is automatically allocated, not upon entry to a procedure, but precisely when a variable becomes used.

The principle of the above, creating new defining-words, could have other uses as well; for example, data for a variable could reside on disk and the code-address of its dictionary entry could point to a routine which retrieves it from disk. What is needed is an easy way to compile these structures. For instance, if we want to create a variable X, to reside on disk, we need another variable, Y, to hold the data, and an operator RETRIEVE; X is then compiled as the definition; Y RETRIEVE so every reference to X returns the value from disk. Having defined the "Y Retrieve" word we need an easy way to tell the compiler that X should be compiled as that kind of definition, without having to write it all out.

PASCAL implementation may be very well optimized, but FORTH code accomplishes even more in the way of code compaction because of its unique representation of operands as though they were operators in object code. This means that no separate operator for "load stack" to push an operand is needed in object code, saving space. This is a kind of software implementation of "tagged memory" on large scale machines. FORTH still requires 16 bits for each object code entry, but further compaction implies what must be a time consuming unpacking operation every time a virtual-machine instruction is executed. The same compaction in FORTH applies to procedure calls as well as to data; no separate "call" op code is needed as the procedure referencing operand is the call itself. Since each procedure

has within itself whatever housekeeping functions are involved with entry and exit, elaborate housekeeping need be performed only when necessary; e.g. locals can have fixed absolute addresses or be dynamically allocated as the case warrants. Have the hard-wired stack-machine designers missed something here?

PASCAL is also capable of making available at run time the power of the compiler for sophisticated interfacing of users to applications packages, by running an interpreter for PASCAL and calling precompiled object code procedures. The interpreter seems large, however, and the whole procedure cumbersome. A high quality APL interpreter will soon be available from MICROSOFT for this kind of appilcation, but APL has many problems running from its special keyboard to the difficulty of modifying it. FORTH thus offers unique advantages in this area.

George B. Lyons
Jersey City, NJ

"The 'TO' Solution" in issue #4 improves the handling of variables by increasing the amount of interpretation done at run-time in FORTH (a conditional branch on the value of the store/retrieve state variable). Perhaps this could also be accomplished instead at compile time. Let each variable contain two code addresses, one for retrieval, one for storing, and let the compiler select which to put into the object code generated on the basis of the state variable. "TO" as such would then not appear in the object code at all, saving space, but variables would become longer. But again, no assignment operators at all would appear in object code, either. In order for the compiler to work this way it would have to be expanded beyond the standard FORTH by adding the capability of distinguishing different types of words and using different procedures to compile variables and non-variables. This might be accomplished by adding to the code for each type of word, e.g. variables, a pointer to a compiler routine to be used when compiling words of each type. When encountering a word in the input stream, it would be looked up in the dictionary, the code address extracted, and then the pointer to the compiler code located within the code; compilation would then continue where pointed. The compile code for variables would check the state variable and enter the appropriate address from the word being compiled. Code for variables would have to make allowance for the varying distance of the parameter field from the code address.

This method might be a technique for implementing compilers of all sorts derived from the FORTH compiler.

In the case of ARRAYS, one might consider having several code addresses in each array in the dictionary, with associated operators in the source code which do not get put into the object code, but merely select which of the code addresses will be compiled for each array reference in source.

Of course, if you expand the compiler, it might get too big to fit in memory with all the application code the way FORTH normally does.

George G. Lyons
Jersey City, NJ

;s

NEW PRODUCTS

REVISED fig-FORTH LISTING

fig-FORTH 8080 Assembly Listing has been revised to Version 1.1. If you have an old version (1.0), send in the original cover and $4.00 for the new version (1.1). Otherwise available for $10.00 (overseas $13.00. Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070.

PDP-11 fig-FORTH DISKETTE

In addition to the fig-FORTH system, the diskette includes an editor, Forth assembler, string package and an 80-page User's Guide with discussion and examples of Forth programming techniques.

Like all the fig-FORTH systems, this one has full length names to 31 characters, the "security package" of compile-time error checks, and alignment with the 1978 Forth International Standard.

This system runs under the RT-11 or RSX-11M operating systems, and can be modified for other environments or for stand-alone It can interface to database packages or other software, allowing interactive access and program development with systems not otherwise available interactively. The fig-FORTH model is distributed in Macro-11 source, for easy modifiability by programmers without a Forth background; the editor, assembler, and string package are in Forth source.

The complete system price is $130, including diskette and all documentation; the User's Guide separately is $20. (Ca. residents add 6% tax.) John S. James, P.O. Box 348, Berkeley, CA 94701.

FORTH FOR THE TRS-80

MMSFORTH offers TRS-80 users stack-oriented logic and structured programming, machine-code speed and compactness, virtual memory, major advantages of interpreter, compiler and assembler (all are co-resident), and your own commands in its extensible dictionary. MMSFORTH includes assembler/ editor, all necessary routines for disk and/or tape, and additional routines for BASIC-like handling of strings, arrays, etc. MMS supplies information and examples to learn this language or to start your own FORTH program for your specialized application.

SPECIAL - At no extra cost, THE GAME OF LIFE in MMSFORTH just 2 seconds per generation, an excellent demonstration of the techniques and speed of FORTH. MMSFORTH, without manual:

L.2 16K tape	$35.00
Disk, w/Disk I/O	$45.00
"The microFORTH PRIMER", manual for MMSFORTH	$15.00

Miller, Microcomputer Services, 61 Lake Shore Road. Natick, MA 01760

SBC-FORTH

SBC-FORTH is a complete firmware operating system designed to plug directly into the ROM sockets of the SBC-80 series of single board computers offered by Intel & National. Operating entirely within the resources of one SBC Card SBC-FORTH features resident compiling and assembling of user entered tasks. The addition of SBC Disk and RAM Cards to the system allows SBC-FORTH's resident disk I/O and Text Editor to edit, load, and run source programs from disk. SBC-FORTH is interactive with the user's CRT and produces tight object code capable of

execution speeds approaching assembler code. SBC-FORTH is presently available on four 2716 EPROMS for the SBC-80/20 and two 2732 EPROMS for the SBC-80/30 CPUs. Options include Single or Double Density SBC Disk I/O Drivers anad CRT Console Baud Rates. Priced from $750 depending on media. Zendex Corporation, 6398 Dougherty Road, Dublin, CA 94566.

STOIC-II

STOIC-II is an enhanced version of STOIC, a FORTH dialect which originated at MIT's Biomedical Engineering Center. STOIC offers significant advantages over other microprocessor FORTH implementations, notably a comprehensive disk file system and text editor in place of the "screen" ap proach used elsewhere. This encourages good program docu mentation by allowing arbitrarily long lines and pages, thus making it easier to include adequate comments and indenta tion. Other features include syntax checking and a separate stack for loop control, which together yield a system that recovers from most errors instead of crashing. STOIC also provides for character-string literals and offers a very extensive set of standard words, including 16 and 32 bit arithmetic.

The standard STOIC-II package includes the kernel and basic words, assembler, file system, editor, double-precision and floating-point arithmetic, target compiler and associated library of primities, and utility programs for paginated file listing and alpabetized listing of vocabulary branches. The version currently distributed runs on 8080, 8085, and Z-80 based computers with soft-sectored floppy disks, and is priced at $4000 with complete source code or $2000 for object code only. Avocet Systems, 804 South State Street, Dover, Delaware 19901.

FORTH, Inc. NEWS

A new on-line monitoring and record-keeping system developed by FORTH, Inc., using a PDP 11/60 minicomputer, is being used by the Department of Pulmonary Medicine of Cedars-Sinai Medical Center, Los Angeles. The FORTH system collects data from five on-line sources, maintains a data base including this information and performs calculations for and formats a wide variety of reports. Data is collected from patient admittance questionnaires and intensive care patient records through terminals; from the automated blood gas laboratory; the pulmonary function test equipment; and the exercise laboratory, all through direct links with the PDP 11/60. The speed of FORTH allows the Cedars-Sinai pulmonary specialists to examine all data on a particular patient, while the patient is still in the laboratory and connected to the equipment.

Current Openings

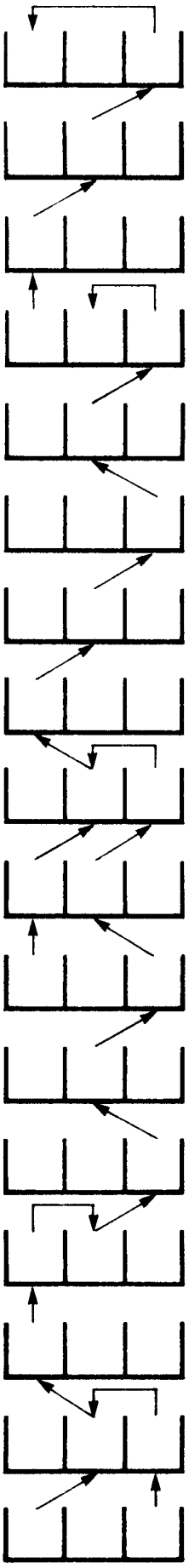
Project Manager - applications and special systems

Product Manager - processors and drivers

Programmer/Engineer - hardware and software projects

Technical Writer - software documentation

FORTH, Inc.
815D Manhattan Avenue
Manhattan Beach, CA 90266



FORTH DIMENSIONS

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

Volume 1
Number 6
Price $2.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume 1 No. 6 March/April 1980

Publisher Roy C. Martens

Editorial Review Board

Bill Ragsdale
Dave Boulton
Kim Harris
John James
George Maverick

FORTH DIMENSIONS solicits editorial material, comments and letters. No responsibility is assumed for accuracy of material submitted. ALL MATERIAL PUBLISHED BY THE FORTH INTEREST GROUP IS IN THE PUBLIC DOMAIN. Information in FORTH DIMENSIONS may be reproduced with credit given to the author and the Forth Interest Group.

Subscription to FORTH DIMENSIONS is free with membership in the Forth Interest Group at $12.00 per year ($15.00 overseas). For membership, change of address and/or to submit material, the address is:

Forth Interest Group
P.O. Box 1105
San Carlos, CA 94070

HISTORICAL PERSPECTIVE

FORTH was created by Mr. Charles H. Moore in 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of dissatisfaction with available programming tools, especially 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.

The Forth Interest Group is centered in Northern California, although our membership of 950 is world-wide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.

PUBLISHER'S COLUMN

This is a special issue of FORTH DIMENSIONS. It is the regular issue but it is also very special. It includes the complete text of Charles Moore's speech at FORTH Convention, October 1979, in San Francisco. The founder of FORTH has given us a historical and futuristic view of FORTH. Thank you, Chuck!

This issue completes Volume 1 of FORTH DIMENSIONS and what a way to finish. The largest issue to date and the complete Charles Moore article. Look for Volume 2, Number 1 soon.

Roy Martens

FORTH, The Last Ten Years and The Next Two Weeks ...

Charles H. Moore
Chairman of the Board
FORTH, Inc.

WELCOME

Thank you. You honor me just by being here and being so involved in something that I never really expected would be this interesting to a group of people. I think way back in the Dark Ages I had in mind maybe some day addressing the Rotarians about FORTH. This is a rather more select group.

It turns out that FIG's estimate of this being the tenth birthday party for FORTH is remarkably accurate. By way of explanation, this is not intended to be a history of FORTH. For one reason, I do not have a very good memory for such events and I am not going to be particularly accurate nor particularly complete -- I am just going to give you my impression of what's happened for ten years. I am not up with what's happening in Europe or even in San Francisco and I apologize for that, but there never seems to be a need to delve into the history of FORTH. There is a history but first I want to talk a little bit about what FORTH is. This has been a subject of some speculation.

ASPECTS

Is FORTH an operating system? Is it a language? Is it a state of mind? I propose to trace five threads of history through ten years. I am going to do it in such an order that if we cut off the end nobody will care.

The five aspects of FORTH are philosophy, language, implementations, computers, and organizations, (meaning groups like FIG). If you talk about FORTH, the language, you can talk about some of these things and if you talk about FORTH the company you can talk about other things. This is a reasonable organization for what is really a broadly based attack upon the problems of society.

When I was very young I don't think I would have liked myself very much. I recollect being rather arrogant — that is a little bit too strong -- I wanted to do things my way, I was not convinced that I should not be permitted to, and I think I was a bit hard to get along with. That's all changed now. But in particular I was insecure. I was promoting certain ideas which everyone told me were wrong and that I thought were right. But if I were right, then all those other people had to be wrong and there were a lot more of them than me. It took a lot of arrogance to persist in the face of rather massive disinterest.

You may have noticed that FORTH is a polarizing concept. It is just like religion or politics, there are people who love it and people who hate it and if you want to start an argument, just say — "Boy, FORTH is really a great language".

I think there were some of you around ten years ago who may be aware of the problems that a programmer would encounter. They are exactly the same problems that a programmer encounters today! There has been no progress in the software industry for the last twenty years. This was apparent ten years ago and it was unsettling. It did not seem that the last thought had been "thunk" when FORTRAN was invented and yet nobody seemed to question that. It was the unspoken assumption that things are the way they are and they cannot become substantially different.

Speech at FORTH Convention, October 1979, San Francisco. CA.

PHILOSOPHY

Let me about philosophy now. I was a free-lance programmer once upon a time (1968). I went to work for a carpet manufacturer and learned COBOL partly out of financial necessity and partly with the thought "here's a language I don't know -- let's pick up one more". These people acquire a graphics system. It was an IBM 1130 with a 2250 graphic display unit, a very nice state-of-the-art outfit, expensive! Speculation was that this would help us design carpets or maybe furniture. Nobody was really sure but they wanted to try. The 1130 was a very important computer. It had the first cartridge disk. It also had a card reader, a card punch, and a console typewriter. The backup for the disk was the card punch! I don't think I ever backed-up the disk but I do remember reloading the operating system numerous times.

The 1130 went away one day because without color it really wasn't worth anything for manufacturing carpets. They also had a Burroughs 5500 which was running ALGOL at a time when ALGOL was not popular. A very nice machine. They were programming in COBOL, which was the first good COBOL. These were progressive people and want to credit them -- Mohasco Industries. I put FORTH on the 5500 -- this is fairly unusual. It was cross-compiled to the 5500 from the 1130, since there is no assembler on the 5500. There is a dialect of ALGOL called ESBOL that Burroughs used to compile operating systems (it was not available to the users). But I learned about push-down stacks from this machine and had a lot of fun. It was third-shift work because the 5500 was busy. It was replaced by a Univac 1108, so I implemented FORTH on the 1108. It controlled the interactions of a bunch of COBOL modules which did all the real work. The 1108 was cancelled due to anticipated financial reverses; the programming staff quit; and I went to work for the National Radio Astronomy Observatory (NRAO).

Before I left -- my last week -- I wrote a book. It was entitled Programming a Problem Oriented Language and it expressed my philosophy at the time. It is very amusing reading. It also describes what was then FORTH. It makes amusing reading because, unknown to anyone, I was expressing opinions, attitudes, not designing a programming language.

FORTH first appeared on that 1130 and it was called FORTH. It had all the essential characteristics of FORTH and I will return to that point later.

F-O-R-T-H is a five letter abbreviation of "fourth," standing for fourth-generation computers. This was the day, you may remember, of third-generation computers and I was going to leapfrog all of that. But FORTH ran on the 1130 which only permitted five-character identifiers.

The first thing that was modern FORTH was the Honeywell 316 program at NRAO. I was hired by George Conant to program a radio telescope data acquisition program. I was given a computer (to the envy of other people who felt they deserved it). I was turned loose to do whatever I could with it, provided I came up with a product. I just went off and nobody really wanted or appreciated what they ended up with.

We developed a number of systems at NRAO and encountered the issue of patenting software. Programs cannot be patented; ought not to be patented; would be very expensive to patent. NRAO has an agreement with Research Corporation, a company that tries to pull from universities some technology spinoffs that can be used to better mankind. (They patent things for people who don't know (or care) how.) FORTH seemed like something that perhaps should be patented, so we spent a year writing proposals, investigating and getting lawyers' opinions. The conclusion was that maybe it could be patented, but it would take Supreme Court action to do it. NRAO wasn't

interested. As inventor, I had fall-back rights but I didn't want to spend $10,000 either, so FORTH was not patented. This probably was a good thing. I think that if any software package would qualify for patenting FORTH would. It has no really innovative ideas in it, yet the package would not otherwise have been put together. If you apply this reasoning to hardware, hardware is patentable. It is one of my disillusionments that the establishment refuses to provide any effective protection for software. Probably it is the lack of vocal objection from within the industry and the willingness to acquiesce, knowing that today's software will be obsolete in a year anyway.

Given interest from other astronomers, a few believers formed FORTH, Inc. We developed miniFORTH (FORTH on minicomputers) with the idea to have a programming tool. The first important realization of that tool came when we put an LSI-11 and FORTH into a suitcase. I think I became the first computer-aided programmer, in that I had my computer and took it around. I talked to my computer, my computer talked to your computer and we could communicate much more efficiently than I could directly. Using this tool we put FORTH on many computers. My goal in all of this was to make myself a more productive programmer. Before all this started, I had figured that in forty years I could write forty programs at the rate I was going. That was it. Period! That was my destiny but I wanted to write more programs than that. There were things out in the world to be done and I wanted to do them.

It has taken a long time. I still don't have the computer I want, but I'm working at ten times that rate and I see other computer-aided programmers now. I am amazed that it should not have been obvious that programmers had to be computer-aided. To expect the programmer to deal with an intrinsically unfriendly machine on his own is not in keeping with the attitude that we preach for other people to follow.

As time went on it became apparent that FORTH is an amplifier. A good programmer can do a fantastic job with FORTH; a bad programmer can do a disastrous one. I have seen very bad FORTH and have been unable to explain to the author why it was bad. There are characteristics of good FORTH: very short definitions and a lot of them. Bad FORTH is one definition per block, big, long, dense. It is quite apparent, but very hard to point to an example of something that went awry or explain why or how. BASIC and FORTRAN are much less sensitive to the quality of the programmer. I was a good FORTRAN programmer. I felt that I was doing the best job possible with FORTRAN and it wasn't much better than what everyone else was doing. I indented things a little more nicely, maybe, and I declared some things that everybody else left to get declared by default. What more can you do? In a sense I said, "let me do it right. Let me use a tool which I appreciate and if everyone can't use this tool well, I am sorry, but that is not my goal." In that sense FORTH is an elitist language. On the other hand, I think that FORTH is a language that a grade-school child can learn to use quite effectively if it's presented in the proper bite-size pieces, with the proper motivation.

Finally, polyFORTH is a condensation of everything we have learned in the last ten years of developing FORTH. I think it is a very good package. I foresee no fundamental changes in the design of the language except for accommodation to the standards which are becoming increasingly important. Up until now there has been no reason for standards. There are internal standards of FORTH, Inc. internal standards at Kitt Peak for the effectiveness of the organization, but there has never been a demand for portability. In

fact I know very few programs that portability has ever been seriously attempted with. The time has clearly come to change that.

There will be developments in other areas and one was brought home most forcibly today. It may be that FORTH is not merely a programming language. It may be saying something much more important about communication — between people, between computers, between animals. This is startling! It had never occurred to me that anyone would really "speak FORTH" in an attempt to communicate with anything else than a computer; it is not any longer clear that that is the case. There may be concepts embodied in FORTH of greater general utility to the basic problem of communication.

Now in concluding the philosophy section, I would like to read a poem. This is a poem that some of you have heard. It is a translation of a classic of English literature and it goes as follows:

```
: SONG
  SIXPENCE !
  BEGIN RYE @ POCKET +! ?FULL END
  24 Ø DO  BLACKBIRD I + @ PIE +! LOOP
  BAKE BEGIN ?OPENED END
  SING DAINTY-DISH KING ! SURPRISE ;
```

The author is Ned Conklin, who is very good at that sort of thing and is the first FORTH poet. Is there a place for this in the world of communication? I don't know. It is remarkably easy to come up with such paraphrasing of just about anything that you care to paraphrase. It's not clear that it's not an efficient means of communication.

INTRODUCTIONS

Let me introduce two people since I have touched upon the subject. It is ten years since there was one FORTH programmer. I would estimate that there are now 1,000 FORTH programmers, which is 2 to the 10th power and comes out nice and round -- a doubling time of one year. Actually, I think the doubling time is slightly shorter than that -- 10 times in three years and that comes out to 2,000 as some people would prefer. What we conclude is that next year there will be twice as many programmers; the year after that twice as many, and if you believe numerology these projections are unarguable. There is a curve, you extrapolate the curve and draw the conclusions. We don't know how it is going to come about. FORTH, Inc. can't train twice as many people next year — well, maybe we can. But, somehow the FORTH community as a whole has got to train twice as many people next year and thereafter. Maybe the Apples and the Radio Shacks are going to be the method of accomplishing that. It seems that capabilities come along just about quickly enough to keep the exponential curve growing. I have fairly great confidence that 1) the doubling time is a year, and 2) it is going to continue. Now there is collateral evidence to support this, if you plot the number of FORTH systems or the dollar value of FORTH systems or percent penetration of markets. Each way, you get about the same growth curve, so I think the growth curve is honest.

Ten years ago there was one FORTH programmer. The second FORTH programmer is in the audience; please meet Elizabeth Rather. Now that is quite a quantum jump, from one to two. The next step was four and they came out of Kitt Peak and the growth can be traced from there, for awhile, if anyone cares to. Actually the first FORTH user is in the audience and that is Ned Conklin. He was head of the station at Kitt Peak for NRAO, running the telescope, responsible for committing his telescope to this risky venture. It is an important telescope because it is responsible for half of the interstellar molecules discovered in the last ten years.

Again, I didn't exactly ask permission to commit these people to this course of action. Nobody realized what the consequences were going to be. It doesn't seem to have worked out too badly. FORTH is still running on that telescope at Kitt Peak and on a lot of other telescopes.

LANGUAGE

Now let's talk about the language and how FORTH came to be what it is today. There is a pre-history which goes back much further than ten years and I have some slides showing that time. These are strictly pre-history -- I found an old pile of listings and I photographed them. The first component of FORTH to occur was the interpreter. [Figure 1] This is an example of an early interpreter programmed in ALGOL. This was done at Stanford Linear Accelerator Center back in the early sixties. This is a program which still exists and it is called TRANSPORT. It designs electron-beam transport systems. You see an early dictionary there. The word ATOM shows the LISP influence. ATOM is an indivisible entity, which we now call a "word." Having read a word DRIFT from an input card, I would execute the drift routine and so on. I have looked through innumerable listings and found this style of programming quite consistent — it's the way I wrote programs in those days. I had an input deck which got interpreted with a structure pretty much as you see it today: words separated by spaces, no particular limits on the length of the words (as you can see from SOLENOID), only the first characters, however, were significant.

FIGURE 1.

```
IF ATOM="DRIFT" THEN DRIFT
ELSE IF ATOM="QUAD" THEN QUAD
ELSE IF ATOM="BEND" THEN BEND
ELSE IF ATOM="FACE" THEN FACE (-1)
ELSE IF ATOM=ROTATE" THEN ROTATE
ELSE IF ATOM="SOLENO" THEN SOLENOID
ELSE IF ATOM="SEX" THEN SEX
ELSE IF ATOM="ACC" THEN ACC

ELSE IF ATOM="MATRIX" THEN BEGIN IF NOT FITTING THEN BEGIN
   REAL A;
   WRITE1(3,0,0,CORE[S]); LINE (-(8+42x(ORDER-1)));
   FOR J=1 STEP 1 UNTIL 6 DO BEGIN
      FOR K=1 STEP 1 UNTIL 6 DO WRITE1(2,8,R1[J,K]xUNIT[K]/UNIT[J1,2);
      LINE(0) END;
   IF ORDER=2 THEN FOR C=1 STEP 1 UNTIL 6 DO BEGIN
```

Here is another example, quite similar. [Figure 2.] Here ATOM has become W and I am looking up + and - and T, R, A and I — which represent an early version of our text editor. That again is ALGOL. I am not completely clear what was being edited. I think it was some kind of files sort program, maybe on cards that were getting printed or rearranged.

FIGURE 2.

```
120 CYCLE; FILL OUTPUT WITH BUFFER[1].BUFFER[2]:
  1 WHILE WORD NEQ "END    " DO
  2 IF W=GM1 THEN REPLY ("OK    ")
  3 ELSE IF NUMERIC THEN L:=MIN(W-1, EOF)
  4 ELSE IF W="+       "THEN L:=MIN(L+WORD.EOF)
  5 ELSE IF W="-       "THEN L:=MAX(L-WORD.O)
  6 ELSE IF W="T       "THEN BEGIN
  7    IF WORD>G 1 THEN W:=1; W:=MIN(L+W-1, EOF);
  8    FOR L:=L STEP 1 UNTIL W DO BEGIN
  9       POSITION: TYPE END; L:=L-1 END
130 ELSE IF W="R       "THEN BEGIN
  1    POSITION; REPLACE END
  2 ELSE IF W="A       "THEN BEGIN
  3    L:=EOF:=EOF+1; REPLACE END
  4 ELSE IF W="I       "OR W="D       "THEN BEGIN
  5    IF NOT RECOPY THEN BEGIN
  6       RECOPY:=TRUE; REWIND(CARD) END;
  7    POSITION; (F W ="I      "THEN BEGIN
  8       PLACE: REPLACE END
  9    ELSE BEGIN EMPTY:=TRUE; IF WORD NEQ GM1 THEN BEGIN
140       L:=MIN(1+W-1, EOF); SPACE(CARD, L-LO+1); LO:=L+1
  1       END END END
```

Here is another way of setting up a dictionary. [Figure 3.] I am filling an array with strings of text and I am going to search that array for a match, take the index and vector through a computed GO-TO.

FIGURE 3.

```
  7 LABEL UNDEFINED, BACKWARD1, TYPE2, FIND, INSERT, DELETE, ERASE,
  8 START, REPEAT1, BOUNDARY1, BEGIN2, END2, QUIT1, ALGOL, FORTRAN,
  9 COBOL, DATA, PACK1;
280 SWITCH SW:=UNDEFINED, BACKWARD1 TYPE2, FIND, INSERT, DELETE, ERASE,
  1 BACKWARD1, TYPE2, FIND, INSERT, DELETE, ERASE, START, REPEAT1, BACK,
  2 BOUNDARY1, BEGIN2, END2, QUIT1, ALGOL, FORTRAN, COBOL, DATA, PACK1;
  3 ALPHA ARRAY COMMAND[1:32];
  4 FILL COMMAND[*] WITH
  5 "-       ", "T       ", "F       ", "I       ", "D       ", "E       ",
  6 "BACKWAR", "TYPE    ", "FIND    ", "INSERT  ", "DELETE  ", "ERASE   ",
  7 "START   ", "REPEAT  ", "        ", "BOUNDAR ", "BEGIN   ", "END     ",
  8 "EXIT    ", "ALGOL   ", "FORTRAN", "COBOL   ", "DATA    ", "PACK    ",
  9 "        ", COUNT:=0; RETURN; COPY ("EDIT@RE" U "ADY    ");
290 TRANSMIT;
  1 BACK: SOURCE(1); WORD1;
  2    IF W="       "THEN GO QUIT1; GO TO SW[MEMBER(COMMAND.W)+1];
```

Here is another way of implementing the dictionary. [Figure 4.] This is the first appearance I have on record of a stack. I am looking up the words in a conditional statement and setting NEXT to the index. And that's the first appearance of NEXT which I can find.

FIGURE 4.

```
  8 PROCEDURE RELEVANCE; BEGIN REAL T,KO;
  9    J:=0; I:=-1; WHILE WORD NEQ "END      "DO
180    IF W="       "THEN NEXT:=3
  1    ELSE IF W="GT     "THEN NEXT:=4
  2    ELSE IF W="LT     "THEN NEXT:=5
  3    ELSE IF W="NOT    "THEN NEXT:=6
  4    ELSE IF W="AND    "THEN NEXT:=7
  5    ELSE IF W="OR     "THEN NEXT:=8
  6    ELSE IF W="+      "THEN NEXT:=9
  7    ELSE IF W="-      "THEN NEXT:=10
  8    ELSE IF W="*      "THEN NEXT:=11
  9    ELSE IF W="/      "THEN NEXT:=12
190    ELSE IF KO:=SEARCH1(W) GEQ O THEN BEGIN
  1       NEXT:=1; NEXT:=k:=KO END
  2    ELSE BEGIN
  3       NEXT:=2;
  4       IF BASE[K|=" " THEN NEXT:=WORDS[0]
  5       ELSE NEXT:= W END;
  6    NEXT:=0 END;
```

Here is the other half of that -- this is the implementation of the stack. [Figure 5.] This is a variant of ALGOL called BALGOL that lets you put assignment statements inside other statements. "Stack of J replaced by J-1" is how you push something onto the stack. One of my "less-liked" features of ALGOL was that I had to play games like "real of boolean of stack of J and boolean of ..." just in order to get around the automatic typing that ALGOL was insisting that I apply. Now this was specifically intended to let me manipulate parameters that were interpreted from the card deck as arguments to the routines. In other words, if I wanted the sine of an angle, I could say ANGLE SINE but if I wanted to convert the angle from one unit to another, I needed at least some simple arithmetic operators and this provided them. This is again at Stanford.

FIGURE 5.

```
   6
   7  BOOLEAN PROCEDURE RELEVANT; BEGIN
   8     I:=J:=-1; STACK[0]:=1; DO CASE NEXT OF BEGIN
   9        J:=-1;
 210        STACK[J:=J+1]:=CONTENT;
   1        STACK[J:=J+1]:=NEXT;
   2        STACK[J:=J-1]:=REAL(STACK[J]=STACK[J+1]);
   3        STACK[J:=J-1]:=REAL(STACK[J] GTR STACK[J+1]);
   4        STACK[J:=J-1]:=REAL(STACK[J] LSS STACK[J+1]);
   5        STACK[J]:=REAL(NOT BOOLEAN(STACK[J]));
   6        STACK[J:=J-1]:=REAL(BOOLEAN(STACK[J]) AND BOOLEAN(STACK[J+1]));
   7        STACK[J:=J-1]:=REAL(BOOLEAN(STACK[J]) OR BOOLEAN(STACK[J+1]));
   8        STACK[J:=J-1]:=STACK[J]+STACK[J+1];
   9        STACK[J:=J-1]:=STACK[J]-STACK[J+1];
 220        STACK[J:=J-1]:=STACK[J]xSTACK[J+1];
   1        STACK[J:=J-1]:=STACK[J]/STACK[J+1];
   2        END UNTIL J LSS 0;
   3     RELEVANT:=BOOLEAN(STACK[0]) END;
   4
```

Now here is a PL/1 program doing very much the same thing at a considerably later date. [Figure 6.] At the top you see JCL (Job Control Language) which was also not a pleasant thing to deal with. One of the criticisms of programming languages that I mentioned in my book was that a programmer at a typical computer center, in order to function, needed to know nineteen languages. This covered writing Fortran programs, submitting card decks, etc. These languages were all subtly different with commas here and spaces there and equal signs meaning different things -- nineteen languages, just to function. Nobody advertised the fact. Nobody sat down and took a course in nineteen languages, but you had to pick them up in the course of several weeks or several months in order to be effective. FORTH, I figured could replace all of them.

Here is NEXT: PROCEDURE CHARACTER. [Figure 7.] I don't remember that syntax but that I think it is the first definition of NEXT as a procedure that went off and got the next word and did something with it. This is still all pre-FORTH. We haven't gotten to what I would consider the first FORTH system.

FIGURES 6 & 7

```
 1  //UTILITY      JOB SYSTEM OVERHEAD
 2  //             EXEC PGM=IEBUPDTE, PARM=NEW
 3  //SYSPRINT     DD   SYSOUT=A
 4  //SYSIN        DD   DATA
 5  ./            ADD  NAME=WORD,LEVEL=00,SOURCE=0,LIST=ALL
 6  NEXT: PROCEDURE CHARACTER(4);
 7     DECLARE KEYBOARD STREAM INPUT, PRINTER STREAM OUTPUT PRINT;
 8     DECLARE (I TEXT CHARACTER (81) INITIAL((81)" "),
 9        2 C(81) CHARACTER(1), I INITIAL(0),W CHARACTER(4),
10        WORD CHARACTER(32) VARYING BASED(),P,NUMERIC BIT(1)) EXTERNAL;
11  P=ADDR(C(I));
12  IF C(1)="-" OR C(I)="." OR "0" LE C(I) THEN BEGIN; NUMERIC="1"B;
13     IF C(I) NOT="." THEN DO I=I+1 BY 1 WHILE "0" LE C(I); END;
14     IF C(I)="." THEN DO I=I+1 BY 1 WHILE "0" LE C(I); END; END;
15  ELSE DO; NUMERIC="0"B;
16     IF "A" LE C(I) THEN DO I=I+1 BY 1 WHILE "A" LE C(I) OR C(I)="-"
17     END; ELSE; I=I+1; END;
18  W=WORD; RETURN(W);
```

Here is a rather later version of FORTH coded for the IBM 360. [Figure 8.] Those are the routines PUSH and POP. PUSH cost 15 microseconds on an IBM 360-50. It includes stack limit checking, which doubled the cost and was one of the things that led me to feel that execution-time stack checking was not desirable and in fact not necessary. However, up to that point, the consequences of a runaway stack were terrifying. POP is there also. It was coded in a macroassembler that did not have stack operations. It was not possible to refer to a previous "anything" so the deck is full of "L19 data constants, address, AL2(*-L18)" to give me a relative jump to the previous one. It could all be done but it wasn't pleasant.

FIGURE 8.

```
830  L18 DC AL2(*-L17)
831      NAME 3,X'+4555)',0  DUP
832+          DC    AL1(3),X'445550 '
833+          DC    X'0'
834+          ORG   *-2-VO
835+          DS    OH
836+          ORG   *+ 0+1
837+          DC    AL1(0*X'40'+X'40').AL2(4)
838  PUSH A SP,MFOUR   COSTS 15 OS
839      ST T,0(,SP)
840      CB SP,DP
841      BCR 2,NEXT  BHR
842         B ABORT
843  L19 DC AL2(*-L18)
844      DC AL1(4),X'4402CF50',X'40',AL2(8)   DPCP
845      LA SP,4(,SP)
846  POP L T,4(,SP)   COSTS 21 US
847      LA SP,4(,SP)
848      C SP, SPOO
849      BCR 12, NEXT  BNHR
850         B ABORT
```

Here is a version of FORTH coded in COBOL. [Figure 9.] This was done at Mohasco, of course. I am setting up a table of identified words which I am going to interpret from an input string. The attitude is so pervasive that I begin to think that I was talking myself into something here. COBOL is fairly difficult to write subroutines for. They have subroutines, they can be performed, but they may not have any parameters. This makes it a little bit awkward to do anything meaningful.

FIGURE 9.

```
 1  MOVE "CONFIGURATION" TO IDENTIFY(4);
 2  MOVE "DATA" TO IDENTIFY(5);
 3  MOVE "FILE" TO IDENTIFY(6);
 4  MOVE "FD" TO IDENTIFY(7);
 5  MOVE "WD" TO IDENTIFY(8);
 6  MOVE "SD" TO IDENTIFY(9);
 7  MOVE "WORKING-STORAGE" TO IDENTIFY(10);
 8  MOVE "CONSTANT" TO IDENTIFY(11);
 9  MOVE "PROCEDURE" TO IDENTIFY(12);
70  MOVE "INPUT-OUTPUT" TO IDENTIFY(13);
```

Here's the first example of FORTH text. [Figure 10.] This came out of Stanford again. The word DEFINE begins (that is,:) a definition and the word END (that is,;) ends it. The "OPEN is obscure. "NAME seems to be the way the name was introduced. Apparently there did not have to be a space between the quote and the word. There are the definitions of a number of stack operators. Top of the line is CODE - "OPEN DEFINE MINUS + END ; I guess that is subtraction. SEAL was an early word for sealing the dictionary for some reason. BREAK, I guess broke the seal. "< : OPEN DEFINE - < END ; is the same definition we use today, in a very early state. That was from a thing I called "Base Two," intended to be some kind of base programming language — I can't remember any more about it.

FIGURE 10.

```
"- "OPEN DEFINE   MINUS  +    END
SEAL   "< "OPEN DEFINE    - <    END BREAK
"NOT "OPEN DEFINE    MINUS 1+    END
"< "OPEN DEFINE   .<    END
"AND "OPEN DEFINE   x    END
"OR "OPEN DEFINE  NOT   .NOT AND NOT    END
   "T 1 1 "REAL DECLARE
"= "OPEN DEFINE   T=; DUP T<  . T> OR NOT   END
"≠ "OPEN DEFINE   = NOT   END
"< "OPEN DEFINE,  > NOT   END
"≤ "OPEN DEFINE   < NOT   END
"DUMP "OPEN DEFINE   NAME 10 "ALPHA WRITE; 3 10 "REAL WRITE 0 LINE  END
```

This is a version of FORTH source for the 5500. [Figure 11.] Again, very early — the second computer that FORTH was put on. Apparently ¢ stands for CODE and these are the code definitions of the stack operation on a 5500. Now the 5500 is a stack machine at a time when stack machines were not at all popular. They did a very good job with their stack. All of these were implemented with one 12-bit instruction and the present names of these operations are directly derived from the names of the 5500 operations. That's where DUP came from, for instance. Notice that the ¢OR was a way of distinguishing the assemblers OR from the FORTH OR before vocabularies were available.

FIGURE 11.

```
LIST
0001  (   'PRIMITIVES'  26 LAST=  30 SIZE=)
0002  ¢ =   S RETURN
0003  ¢ @  <SD RETURN
0004  ¢ +V 241, RETURN
0005
0006  ¢ OR   ¢OR RETURN
0007  ¢ AND  ¢AND RETURN
0008  ¢ NOT  115, RETURN
0009  ¢ DUP  ¢DUP  RETURN
000A  ¢ SWAP ¢SWAP  RETURN
000B  ¢ DROP ¢DROP  RETURN
000C  ¢ +  +1  RETURN
000D  ¢ -  -1  RETURN
000E  ¢ MINUS  ¢MINUS  RETURN
000F  ¢ *  *1  RETURN
0010  ¢ /  /1  RETURN
0011  ¢ MOD ¢MOD  RETURN
```

Here's an example of FIND coded for the 5500. [Figure 12.] Notice that the word SCRAMBLE is referred to, which is a colon-definition for doing a hashed search. Apparently here I had eight threads, just as we put in polyFORTH last year. These ideas go way, way back. This is FORTH after the threshold was crossed, ten years ago, almost exactly. One can become a little bit depressed at the "tremendous" rate of progress in the last ten years when you see that it was all back there.

FIGURE 12.

```
0013  ¢SM ¢FIND   SCRAMBLE  <SD  ¢DUP
0014     41 >A  41 >B  ¢BEGIN  V  <J  1771,  ¢IF
0015       ¢BEGIN  VO <U  1771,  ¢IF
0016          1 <L  RESULT
0017        ¢THEN   ADDR  ¢DUP  1 <L  <S
0018        US WORD  <J  ¢EQUAL  ¢IF
0019          V1  _J  US  RESULT
001A        ¢THEN  ¢DUP  <SD  ¢BACK
001B     ¢THEN  GET  ¢BACK
001C  :  FIND  TOP  ¢FIND  ¢IF  UR <UD ¢B  ¢THEN;
```

Here is another example of source. [Figure 13.] This was from the Univac 1108. These are very early record descriptions. This is the layout of a record in a file with the name of the field and the number of bytes in the field. That was the Dun & Bradstreet reference file for looking up bad debts.

FIGURE 13.

```
3 DBI    DBI/MCORE 33 33
4      DUNS 8  NAME 24  STREET 19 CITY 15 STATE 4 ZIP 5
5      PHONE 10  BORN 3  PRODUCT 19  OFFICER 24  SIC 4 SIC1  4  SIC2
6      SIC3 4 SIC4 4 SIC5 4  TOTAL 5.0 EMPL 5.0 WORTH 9.0 SALES 9.0
7      SUBS 1 HDQ 1 HEAD 8 PARENT 8 MAIL 19 CITY 15 STATE 14
8      NAME1 19
9END
```

This is the last slide. [Figure 14.] This is a modern FORTH source from FORTH, Inc. Vector arithmetic coded for the Intel 8086. CODE V+ with comments listing the arguments that are on the stack with the results being obvious. Fairly nicely spaced out code. It just looks like good, clean pure FORTH instead of that other gibberish. Not at all cryptic, perfectly obvious to the discerning reader and that's the end of the slides.

FIGURE 14.

```
0     (VECTOR ARITHMETIC)
1  CODE V+ ( Y X  Y X)  0 POP   1 POP   2 POP   3 POP
2     2 0 ADD  3 1 ADD  1 PUSH  0 PUSH    NEXT
3  CODE V- ( Y X  Y X)  2 POP   3 POP   0 POP   1 POP
4     2 0 SUB  3 1 SUB  1 PUSH  0 PUSH    NEXT
5
6  CODE UMINUS  0 POP  1 POP   1 NEG   1 PUSH
7     0 NEG   0 PUSH  NEXT
8  : UMIN   ROT MIN >R MIN R> ;
9
10 CODE V*/   ( Y X M D)  7 POP   3 POP   1 POP   0 POP
11        3 IMUL   7 IDIV  0 PUSH
12     1 0 MOV   3 IMUL  7 IDIV   0 PUSH   NEXT
13
14
15
COPYRIGHT FORTH, INC  OCT. 1979
```

Let me now sketch in time sequence the operations that were implied by some of those slides. The first thing to exist was the text interpreter reading punch cards. The next thing to exist was the data stack with the manipulations of the operators. Those took place very early, around 1960.

Nothing substantial then happened until 1968 and that was the 1130 and the ability for the first time to totally control the way the computer interacted with the programmer. This machine had the first console I had ever seen. I had always submitted card decks -- now I had a typewriter. Do you know what I did? I submitted card decks. It took a long time to figure out how to use the keyboard and whether or not the keyboard would do anything for me. I was very good at a keypunch and I really didn't care if there was a console or not.

The first FORTH was coded in FORTRAN. Very shortly thereafter it was recoded in assembler. Very much later it was coded in FORTH. It took a long time to feel that FORTH was complete enough to code itself. The first thing to be added to what had already existed was the return stack and I don't remember why that was. I don't remember why I didn't just put the return information on the parameter stack. It was an important development to recognize that there had to be two stacks -- exactly two stacks, no more, no less.

The next thing to be added was even more important. I don't know if you appreciate it but it was the invention of the dictionary. In particular, the dictionary in the form of a linked list. In particular, the existence of the code field in the header. For control, up until then, flags had been set, computed GO-TOs executed, some mechanism for associating a subroutine with a word. Now the existence of the address of the routine (rather than an index to the routine) made an incredibly fast way of executing a word once it had been identified. No other language has a code field or anything resembling it. No other language feels obliged to quickly implement the code that the word identifies. You can go about it at your own precious time, but even in these days it was important that FORTH be efficient. The whole purpose of this system was to draw pictures on the 2250 display. The 2250 was a stand-alone minicomputer interfaced with the 1130. What came out of the 1130 was a cross-assembler which assembled the instructions which were then to be executed by the 2250. I think the 2250 had its own memory. Again, very sophisticated things being done very early in a very demanding environment. IBM software, in 16K of memory, could draw pictures on the 2250 fairly slowly. What I accomplished in 4K would draw three-dimensional moving pictures on the 2250. But it could

only do that if every cycle were accounted for and if the utmost was squeezed out -- that's why FORTRAN had to go. I couldn't do an impressive enough job with FORTRAN, an assembler was the requirement.

At this time colon definitions were not compiled -- the compiler came much later. The text was stored in the body of the definition and the text interpreter reinterpreted the text in order to discover what to do. This kind of contradicts the efficiency of the language but I had big words that put up pictures and I didn't have to interpret too much. The cleverness was limited to squeezing out extraneous blanks as a compression medium and I am told that this is the way that BASIC executes today in many instances.

This machine had a disk -- I can't prove it but I am almost certain that the word BLOCK existed in order to access records off the disk. I do remember that I had to use the FORTRAN I/O package and it wouldn't put the blocks where I wanted them -- it put the blocks where it wanted them and I had to pick them up and move them into my buffers. It had an assembler for assembling 1130 code, it had a target assembler for assembling 2250 code and clearly B-5500 code. The B-5500 program was taken far enough to recompile itself. Beyond that there was no application because there was no way I was going to get access to that machine outside of the third shift.

This was the transition point between something that could not be called FORTH and something that could. All the essential features except the compiler were present in 1968. It took a long time for the next step.

The first compiler occurred on the Honeywell 316 system at NRAO several years later. It resulted from the recognition that, rather than reinterpreting text, the words could be compiled and an average of five characters per word could be replaced by two bytes per word at a compression of a factor of two or three. Execution speed would be vastly faster. Again, if it was that easy, why hadn't anyone else done that? It took me a long time to convince myself that you could compile anything and everything. The conditional expression, for instance, had to be compiled somehow. Before compilation, if you came to an IF you could scan ahead in a text string until you came either to an ELSE or a THEN. How did you do an IF if you were going to compile things? But, it worked! Again, I can't remember the sequence.

It may be that the 316 compiled the 360, but I think the 360 compiled the 316. Again, in the early days of FORTH, the idea of of today was there -- cross compiling, cross assembling between different computers was there. Interrupts came at about this time. It was important to utilize the interrupt capability of the computer but it had not been done by me before that. I didn't know anything about interrupts, but I/O was not interrupt-driven. Interrupts were available for the application if it wanted them. FORTH didn't bother.

The multi-programmer came along a couple of years later when we put an improved version of the system into the Kitt Peak PDP-11. This multiprogrammer had four tasks. Input was still not interrupt-driven, which was unfortunate. Interrupt-driven I/O came along when FORTH, Inc. produced its first multi-terminal system. It did not speed things up particularly. If you count cycles it was much more efficient, and it prevented any loss of characters when many people were typing at the same time. FORTH didn't have to look quickly to get each character before the next one came along, as they were all buffered and waiting.

Data-base management came along at this time. It has been extensively changed, just like FORTH has, but

fundamentally nothing has changed. The concept of files and records and fields that I outlined this afternoon dates back from 1974 or so.

The first target compilers came along later with microFORTH. They are very complex things, much more so than I had expected them to be.

I think that completes the capabilities that I think of as FORTH today and I think you can see how they dribbled in. At no point did I sit down to design a programming language. I solved the problems as they arose. When demands for improved performance came along I would sit and worry and come up with a way of providing improved performance. It is not clear that the process has ended, but I think it is clear that that process has now got to be carried into the hardware realm.

HARDWARE

I have designed some computers. This is expensive because I am supposed to be earning money by writing software. I think that hardware today is in the same shape as software was 20 years ago. No offense, but it's time that the hardware people learned something about software and there is an order or two magnitude improvement in performance possible with existing technology. We do not need pico-second computers to make substantial, really substantial, improvements in speed. And faced with that realization, there is no point in trying to optimize the software any further until we have taken the first crack at the hardware. The hardware redesign has got to be as complete as the software redesign was. The standard microprocessors did not have FORTH in mind. Those minicomputers that can be microprogrammed cannot be microprogrammed well enough to even be worth doing. The improvements available are much greater than you can achieve by these half measures.

IMPLEMENTATIONS

All right, let's switch gears a little bit. I would like to talk about the implementations of FORTH of which I am aware. I have touched on them already but I want to rattle off a string of CPUs which have come to mind just to dazzle you with the capabilities. It is actually a tour through the history of computers and it is fascinating that this could all have happened in ten years.

FORTH has been programmed in FORTRAN and in ALGOL and in PL/1 and in COBOL and in assembler and in FORTH. I am sure some of you can come up with other languages with the same history. It has been done on the IBM 1130, the Burroughs 5500, the Univac 1108, the Honeywell 316, the IBM 360, the Nova, the HP 2100 (not by me, but by Paul Scott at Kitt Peak), the PDP-10 and PDP-11 (by Marty Ewing at Cal-Tech), the PDP-11 (by FORTH Inc.), the Varian 620, the Mod-Comp II, the GA/SPC-16, the CDC 6400 (by Kitt Peak), the PDP-8, the Computer Automation LSI-4, the RCA 1802, the Interdata, the Motorola 6800, the Intel 8080, The Intel 8086, the TI 9900 and coming soon the 68000, the Z8000, the 6809 -- I know you people have 6502s and Four Phase. (Audience: And Illiac!) I've raised the question -- is it the case that FORTH has been put on every computer that exists?

COMPUTERS

We speak now about FORTH computers — there are FORTH computers. The first one I know of was built at Jodrell Bank in England around 1973. It is a redesign of a Ferranti computer that I think went out of production. They were going to build their own bit-slice version and they discovered FORTH about the same time, modified the instruction set to accommodate FORTH, and built what I am told is a very fast FORTH computer. I have never seen it. I

have talked to its designer, John Davies, who is one of the early FORTH enthusiasts and eminently competent to do this.

In 1973 came General Logic and Dean Sanderson. The machine qualifies as a FORTH computer because it has a FORTH instruction set and there is a story there. Dean showed me his instruction set and there was this funny instruction that I couldn't see any reason for. I figured it was some kind of no-op or catch-all because it had the weirdest properties. It couldn't possibly be useful -- it was NEXT. It was a one-instruction NEXT -- it was beautiful. And it was a very simple modification to the instruction set. A few wires here and there and that was the first time I saw a FORTH computer. Here was the ability to change an ordinary computer to make it into a FORTH computer.

We have had some ideas about other such modifications that could be made effectively, but I don't know that any of them have been carried out. I am told that Cybek has got its own machine now, built by Eric Fry. These are rumors that I'm just passing on. Child, Inc. does raster graphics systems. I am told they were working on a FORTH computer that was supposed to be available last February -- I haven't heard. Again, I think they are competent to do it and do it well. A blindingly fast FORTH computer on a board, probably biased towards graphics applications, raster graphics.

I have built a FORTH computer called BLUE. It's small. It has never executed any FORTH yet. The design changes as fast as the chips can be plugged into the board, but it's not hard to do.

What are the characteristics of a FORTH computer? It does not need a lot of memory. 16K bytes is about right. Half PROM, half RAM, maybe. It does not need a lot of I/O ports -- it does not need any I/O ports except for the application requirements. A serial line is nice; a disk port is nice. We have put FORTH on an 8080 with disk replaced by enough core to hold 8 blocks. Quite viable, no particular problem with system crashes, a somewhat protected environment. Bubble memories are coming, and Winchester drives of course. We don't need much mass memory, we only need on the order of 100 to 250 blocks. The fact that FORTH can exist quite happily on a very small machine by contemporary standards should be exploited.

ORGANIZATION

Finally, I would like to run through the history of the organizations which have been involved with FORTH. They form another thread to the tapestry. Mohasco, of course, and the National Radio Astronomy Observatory. They birthed it and they rejected it. Thats pretty much why I am here today instead of in South America programming telescopes. They had what we have learned to identify as the NIH Syndrome but in a weird mutated case because it was invented there! It is their loss. You have perhaps read about the VLA, a very large array of antennas in New Mexico -- a very exciting project. Something that I really would have liked to have programmed. It wasn't in the cards (and they have software problems today). On the other hand, there was Kitt Peak. Astronomers are a conservative lot. This may be surprising. I think they are surpassed only by nuclear physicists in being conservative. We have been unable to scratch the nuclear physics field here although I am told that CERN is interested. NRAO is a sister laboratory to Brookhaven. One would think that there would be some communication between them and there isn't. They are both managed by the same University Association. We couldn't interest Brookhaven and we couldn't interest NRAO, but we could interest Kitt Peak and Elizabeth Rather

is the one who did it. She liked FORTH and talked a lot of other people into liking it, too. Kitt Peak adopted FORTH -- gave it the impetus because Kitt Peak is the show place of the astronomy world. Kitt Peak put FORTH on a whole batch of Varians. It's fun because I remember Varians breaking down and spares being wheeled in -- there were 14 of them lined up in the hall. Reliability through redundancy? Pipeline architecture? A lot of other observatories picked it up.

We were deluged by requests for FORTH systems from astronomers and went into business to try to exploit that market. It is a market we would still be in today except that there are so few new telescopes in the world, and you can't support a company on that market.

The formation of FORTH, Inc. was important because I don't think we would be here today if it weren't for FORTH, Inc. We worked very hard to try to sell this thing. We didn't know what we were getting into; we were your classic naive, small business folk. I caution any of you against thinking it is easy to go into business for yourself. It is fun, but the advice is true -- do not go into an area where you must create the demand for your product. But that was the least of the problems really that FORTH has faced. The next step was probably DECUS. Marty Ewing gave his PDP-11 FORTH system to DECUS. I didn't know if that was a good idea at the time -- free FORTHs floating around. It was important because a lot of people were exposed to FORTH who otherwise would not have been. I imagine we picked up a few sales through that channel. Cybek came along. Cybek is probably the savior of FORTH, Inc., in that it provided us lucrative business at a time where we desperately needed it to stay alive. Art Gravina, the President of Cybek is the one that designed (if that's the word) our data-base management system. We can argue about who did what, but Art provided the opportunity to do commercial systems. He got a good deal because he could handle 10 times as many terminals as he could with the BASIC program that preceded FORTH; we learned everything we know about data base management from him. We also acquired our distaste for commercial programming at that time. I commend those of you who are involved in it. I find it much to heavy in system analysis for my taste. You've got to go in there and tell the businessman what he has to do, talk him into it, and hold his hand all through the process of installation. It is a different talent from that of writing programs. Don't underestimate the cost of support.

I think that is about the time that the International Astronomical Union met and agreed on FORTH as a standard language. That was a boost in the world of astronomy although the world of astronomy was no longer the major driving force in the popularity of FORTH. I think EFUG came along about that time -- this was '76 or so -- the European FORTH Users Group. It turned out, to our surprise, that Europe was a hotbed of FORTH activity of which we were largely unaware and perhaps really still are unaware, in that we are not involved in that world and don't quite appreciate the level of interest. FST (FORTH Standards Team) probably began in EFUG's first meetings. Later, a couple of years ago, FIG was started and now we have FORML (FORTH Modification Laboratory), which is an idea-generating organization. The tendency seems to be for people to organize themselves into groups. Some of these groups are companies, some of these groups are associations. It looks like FORTH is going to be a communal activity in the sense of unstructured clusterings of like minded people. The suggestion is that this whole world of FORTH is going to be quite disorganized, uncentralized,

uncontrollable. It's not bad, it's perhaps good.

CONCLUSION

To close on a philosophical note: power to the people. This is the first language that has come up from the grassroots. It is the first language that has been honed against the rock of experience before being cast into bronze. I hesitate to say it is perfect. I will say that if you take anything away from FORTH, then it isn't FORTH any longer, that the basic components that we know are all essential to the viability of the language. If you don't have mass memory, you've got a problem and it can't be waved away. I hesitate to predict. I don't know what is going to happen. I think my view of the future is more unsettled tonight than it has been for years. Promising, yes, confusing, and perplexing. The implications are perhaps as staggering now as they were ten years ago. The promise of realization is much higher. This is ten years of FORTH.

My original goal was to write more than 40 programs in my life. It think I have increased my throughput by a factor of 10. I don't think that that throughput is program-language limited any longer, so I have accomplished what I set out to do. I have a tool that is very effective in my hands -- it seems that it is very effective in others' hands as well. I am happy and proud that this is true.

I wish that the future smiles on you and all of your endeavors.

(Mr. Moore's address concluded to an extended standing ovation.)

DISCUSSION PERIOD

Question: When did <BUILDS and DOES> come along?

Mr. Moore: That's a good question -- ;CODE came first. I think it began way back in 1130 days with the notion that you could define a word that would define other words. That was staggering. I couldn't grasp the implications. ;CODE was a very esoteric word. I explained it to people proudly but I couldn't express the potential I saw in it. I didn't know what ;CODE should do. (It specified the code to be executed for a previously defined word.) I don't have it but I think the initial assembler code for ;CODE was three or four lines long. One of the driving forces behind the address interpreter was making it possible to code ;CODE cleanly. This had all kinds of implications as to what registers should be available. W should be saved in a register instead of (pardon the expression) direct threaded code recovered somewhere because that was expensive. I had a lot of trouble with ;CODE. That was the most complicated routine I had coded in this systems programming fashion. Not so much later it seemed that there ought to be an analog of ;CODE which specified the code to be interpreted when you executed a word. It seemed the natural balance, but I hadn't the foggest idea of what the implementation should be. The first definitions of ;: required three or four lines of code. You had to do what ;CODE did and then more and this couldn't be explained to anyone. Out of that grew the distinction between compile time action and execute-time action and the present form of CREATE and DOES>. This was due to Dean Sanderson, again. It was very convenient for words to be coded to act this way, but it was expensive. It required not only the address of the code to be executed, but the address of the code to be interpreted as well as the parameter to be supplied to the code to be interpreted. Questions included: should the parameter be put on the stack or should the address of that parameter be put on the stack? Should that parameter be at the

beginning of the parameter field where it was a little bit awkward or one word in? Now I don't know if you are aware of the new DOES>? It now has full symmetry with ;CODE.

Again, this grew out of a clear perception of what the word is and what it does. I know of no way of speeding the process from initial thought to development except to let a certain amount of time pass. We sat and debated this thing endlessly and missed the obvious. The current implementation of DOES> does not require the address of the code to be interpreted. That is supplied by a different mechanism and therefore the parameter can occupy the parameter field as it is supposed to. Therefore you can "tick" it and change its value, which is wonderful, except that it is going to be in ROM and won't matter. We save two bytes per DOES> definition. Two bytes per word for a very common class of words and for three years we didn't realize that we had missed the optimum by so much! Although this is proprietary to FORTH, Inc., I am sure that given these clues all of you will proceed to go off and invent the new DOES>. Maybe that's the way things should be.

Question: When you were in the philosophy section, you said that you still don't have the computer that you want and you sort of alluded to that at various times. Can you tell us just what computer you would like to have?

Mr. Moore: Well, first and foremost it has to be reliable. I want an MTBF of ten years. I see around me computers that fail in six months and that is preposterous. You do not get an MTBF of ten years by taking ten computers with an MTBF of six months and lining them up in parallel. I want a computer that I can drop in the ocean and fish out and it doesn't care. I want a computer without an on/off switch. I want it to be small. By small I guess I mean I want it to fit in my pocket, I don't really want it on my wrist. These considerations have consequences. I consider that a reliable computer is one with small parts count. The probability of failure is proportional to the number of parts. So if you only have six parts you can last maybe ten years -- I don't know. I would like it to have voice input/output. A terminal is acceptable but this isn't small. Are you familiar with the Write-Hander? A very nice input device which, if it's perfectly matched to your fingers and if you can train your reflexes to depress two keys at the same time, is a good idea. I am afraid the implementation is not perfect. It probably has to be customized to your hands somehow and even then it is only a substitute for voice input. No power supply, of course, it should run off body heat.

The potentials in the field of communication are enormous. I don't like telephones much. I speculate that I would talk on the telephone if there were a computer between me and it. I speculate that a personal computer like this could be an interface between an individual and society. If I wanted a new driver's license I would tell my computer to get me a new driver's license and it would deal with the bureaucracy.

Question: Can you develop a version of FORTH that will be machine-independent?

Mr. Moore: The premise is wrong. The equivalence of FORTH on different machines requires meticulous attention to the characteristics of these machines. You must use all the hardware capabilities of each machine and you must then work to force it into the mold specified by FORTH's virtual machine.

The internal characteristics of every machine can and must be exploited. You do not need any particular number of registers or stacks, as they can all be simulated; but if

you neglect the capability of the machine, you can end up a factor of two down from where you might otherwise be.

Question: So far I've heard you say that things can be done differently but I wanted to hear what you had to say about the architecture of the processor itself.

Question: I would like to support that question with one more question. I have a customer who ran 64 users on a 5500 for ten years and bought a VAX and can only run 16. My question is -- which is progress and what is a suitable architecture for a FORTH machine?

Mr. Moore: If you measure the size of those two machines, we could put 64 users on a VAX, no problem. The fact that he couldn't is a condemnation of the software, not the hardware.

The characteristics of a FORTH processor? I don't want to go into detail but I will say it is substantially simpler than machines like the 8086, 6800s, simpler than the 8080, probably. You don't need much more than the ability to execute microcode. Very simple microcode will serve for this very simple architecture and the performance comes out of the simplicity. If you add complexity, you are going to decrease your reliability and increase power consumption, all those bad things. I would like to see some studies made of the tradeoffs. I would like to know how complicated FORTH is. Take FORTRAN. You cannot measure its complexity. You can't say a FORTRAN program is 10 to the sixth power in complexity. You can't measure that, you can't separate FORTRAN from the operating system, from the floating-point hardware. You can't determine how complicated a problem it is that that FORTRAN is trying to solve. If we had a standard FORTH implemented on a chip with enough RAM, PROM, and CPU and all FORTH words on that chip, we could measure the area of that chip and get a measure of the difficulty of the task that we were trying to solve. We could compare two implementations with different hardware/ software tradeoffs based on the common measure of the area it took to implement. I think that would be a very, very interesting thing to do.

There is a speculation that human brains have a great capacity, 10 to the 13th bits, some preposterous number. I suspect that human brains have a much smaller capacity than that, on the order of 10 to the 9th bits maybe. We make up for our bad memories by "faking it" a lot; people don't remember what happened ten years ago. I reconstruct what must have happened based on little clues that I pick up here and there. Those reconstructions are so accurate and so impossible to contradict that we get the impression that we can remember a very great deal. I suspect that we can build a computer as complicated as a human brain now and it won't need to be powered by Niagara Falls. It will be a small box.

Question: Assuming someone were to design a FORTH small micro-engine of some sort, how much faster or more capable do you think it would be over the present implementations? Just assuming that someone has done that. No guesstimate?

Mr. Moore: No, and I am not saying and don't you say either, Dean.

Question: Can we assume that it is less than 100 and more than 1?

Mr. Moore: A substantial improvement. We've got to have some edge. The rest of the world is going to come up and clobber us.

Question: You mentioned before the philosophy of seeking the least complicated solution for a problem. Would you comment on simplicity versus flexibility for a solution. How do you judge, how do you make tradeoffs?

Mr. Moore: If you have a thing that does one thing well it is of no interest or value unless the one thing that does well is FORTH. I have no idea. It never occurred to me to ask how you measure flexibility.

Question: Could you extrapolate present trends to the future?

Mr. Moore: I think the obvious extrapolation is telepathy. That in 20 years we will have the functional equivalent of telepathy. If you want to talk to anyone in the world you will be able to do so with minimal apparatus and you will be able to talk to them in the sense of not intruding upon them as telephones do now, but rather relaying messages from your computer to their computer and holding vast dialogues in the way computer mediated conferencing is being done today through terminals. This will be a very natural way of conducting our lives. It requires nothing in the way of technological breakthroughs! It merely wants the implementation of optical communication links worldwide.

I'm reluctant to quote science-fiction books, but there was one recently dealing with the first manned Mars mission and the fact that the United States came up with project management techniques of an advanced order through the use of computers — and completely outstripped the rest of the world. We gained an ascendant position in the world and earned the hatred of the rest of the world for our superiority, tried desperately to export technology and failed. The same thing can happen here. If we have a very tightly integrated community in terms of human resources and the rest of the world is excluded, we are going to have one helluva problem. I think this will happen. I think it will happen without any deliberate planning, and I think it is going to cause problems as great as those we have today between the haves and have-nots. But it will probably also generate the solution.

As to the twists that might occur, there are two wild speculations. One, that computers become intelligent or aware. Nobody has a clue as to what that means, but it is conceivable. Two, that we learn to record human personalities in machines. That merely requires the ability to detect and record the requisite volume of information which, as I say, I don't think is all that great.

Question: I wonder if you care to comment about the possibility of a new direction of architecture having to do with associative memory, which the brain seems to have.

Mr. Moore: I've studied these associative memories: there is such a chip on the market. I'd like to say they ought to be useful in implementing a FORTH computer but, I'm afraid I'm stuck in the mold. When I first encountered LISP I couldn't conceive of a language that didn't have a store operator. How could you do anything if you couldn't store your results somewhere? I can't conceive of a piece of data that is its own label, if you will. I am so used to thinking of data having addresses that my mind just doesn't grasp the possibilities if they don't. So, I'm afraid I can't say anything useful about associative memories.

(The evening concluded with an extended ovation.)

Convention Transcriptions

Audio tape transcripts of the FORTH Convention held in San Francisco, October, 1979 are available. FIG member Jim Berkey recorded the technical sessions and the banquet speech by Charles Moore.

Our thanks to Jim for this significant contribution to FORTH history.

There are four tapes and they may be ordered at $4.00 each, postpaid from: Audio Village, PO Box 291, Bloomington, IA 47402.

Tape 1 - Bill Ragsdale, Welcome and Introductions; Standard Teams Report; General Announcements; Case Statement Contest.

Tape 2 - More Standards Team Reports; FORML; Applications of FORTH: Language Concepts.

Tape 3 - Continuation of Language Concepts; Technical Workshop with Charles Moore.

Tape 4 - Banquet Speech by Charles Moore, "FORTH, The Last Ten Years and The Next Two Weeks." (Printed in this issue of FORTH DIMENSIONS.)

Manuals

Here's an update of FORTH manuals currently in print and commercially availabe i.e. not part of licensed material.

...Using FORTH, 1979, Carolyn Rosenberg and Elizabeth Rather, 160 pp, $25.00. Order from FIG or FORTH, Inc., 2309 Pacific Coast Highway, Hermosa Beach, CA 90254.

...microFORTH™ Primer, 2nd Edition, 1978, 60 pp plus glossary. Order from FORTH, Inc. or Miller Microcomputing Services, 61 Lake Shore Road, Natick, MA 01760.

...Kitt Peak Primer, Feb. 1979, Richard Stevens, 200 pp plus glossary. Order from FIG.

...Introduction to STOIC, Mar. 1978, Jonathon Sachs. Order from Stephen K. Burns, MIT, Room 20A-119, Cambridge, MA 02139 or Wink Seville (714) 452-0101.

...Cal Tech FORTH Manual, June 1978, M.S. Ewing. Send $6.00 to Cal Tech Bookstore, 1-51, California Institute of Technology, Pasadena, CA 91125.

...URTH Tutorial, University of Rochester. Send $20.00 to Software Ventures, 53 Arvine Heights, Rochester, NY 14611.

...FORTH Introduction Reprints, 38 pp, summary and collection of published and unpublished articles about FORTH. Send $10.00 to John S. James, P.O. Box 348, Berkeley, CA 94701.

Editor...

Know anymore? Send us your lists.

MEETING NOTICES

NORTHERN CALIFORNIA

FIG Monthly meetings are held the fourth Saturday of each month at the Special Events Room of the Liberty House department store in Hayward. Informal lunch at noon in the store restaurant, followed by the 1:00 p.m. meeting. Directions: Southland Shopping Center off Highway 17 at Winton Avenue in Hayward, CA. Third floor rear of the Liberty House. Dates: 3/22/80, 4/26/80, etc. All welcome.

THREE NEW GROUPS

FORTH Ottawa Group
c/o W. Mitchell
39 Rockfield Crescent
Nepeah, Ontario, K2E 5L6, CANADA

FORTH UK Group
c/o William H. Powell
16 Vantorts Road
Sawbridgeworth, Herts
CM21 9NB, ENGLAND

Mr. Edward J. Murray
Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001, Union of South Africa

Congratulations! Let us know what you're doing.

People who want to organize local groups can write to FIG for organizational aids and names of other members in your areas. Start a group!

MASSACHUSETTS

Dick Miller of Miller Microcomputer Services announces monthly meetings of the MMSFORTH Users Group. Meetings are on the third Wednesday of the month at 7:00 pm in Cochituate, Mass. Call Dick at (617) 653-6136 for the site and more information.

Incidentally, Dick offers a TRS-80 FORTH System and Z80 and 8080 assembler, data base manager and floating point math extensions. Enthusiastic comments have been received.

FORTH DAY AT NCC

Tuesday, May 20, is FORTH DAY at the PERSONAL COMPUTING FESTIVAL at the National Computer Conference which is being held at the DISNEYLAND HOTEL, ANAHEIM, CA on May 19-22. FIG members wishing to break out their portable soapbox are invited to participate. Speakers may address the audience on any FORTH related topic or may be part of a panel discussion. FIG will have a table in the exhibit area for distribution of FIG literature and individual discussions. Figgers with running FORTH systems with a good demonstration package may show off their efforts at the exhibition area. Members with a paper may present it and have it become a part of the proceedings. FIG has committed to this effort on a short deadline. Those wishing to participate should contact Jim Flornoy (415) 471-1762 immediately. There will also be a FIG meeting at the show.

FIG DOINGS

FIFTH COMPUTER FAIRE

Look for FIG at the Fifth West Coast Computer Faire! We'll be at Booth 1028 in Brooks Hall, San Francisco (the lower exhibit area), on March 14-16. We'll also have a Users Group Meeting on Saturday. All FIG Publications will be available. This will be an ideal time to sign up for FORTH DIMENSIONS, Volume II.

The Booth and meeting site are through the generous consideration of Jim Warren and the Faire management. Thanks again!

FIG MEMBER PAPER

Joel Shprentz of the Software Farm, Reston, VA, will present a paper at the Fifth Computer Faire entitled "Solving the Shooting Stars Puzzle". Joel sells TRS-80 tiny-FORTH and features a FORTH solution to this network problem, using a modification of Dijkstra's algorithm for the shortest path problem.

OLD NEWS BUT INTERESTING

The West Coast Computer Faire is one of the best personal computing conferences and exhibitions. The Fourth Computer Faire was held in San Francisco on May 11-13, 1979 with more than 14,000 people attending.

FIG had a booth where it released the first of the public domain fig-FORTH implementations. FIG also sponsored a FORTH Users Meeting at which about 100 people attended to hear seven vendors talk and answer questions about FORTH systems.

A four hour technical session was also held on the topics of FORTH introduction, Extensibility, Standards, FIG implementation, poly-FORTH, multi-tasking in URTH, and ARPS. The conference proceedings are available for $14.78 from Computer Faire, 333 Swett Road, Woodside, CA 94062.

EUROPEAN MEETINGS

Promptly after the Fourth West Coast Computer Faire, several FIG members, including Kim Harris, Bill and Anne Ragsdale, John James and John Bumgarner, sallied forth (sic) to Amsterdam for the annual European FORTH User's Group Meeting, May 1979.

The meeting was held at the State University at Utrecht, under the hospitality of Dr. Hans Nieuwenhuizen. In addition to papers from FIG members that were also presented at the Computer Faire, there were status reports on standards efforts and reviews of European developments.

The next meeting is planned for Nancy, France, hosted by TECNA. Incidentally, TECNA is one of the most technically progressive firms in using FORTH as a base for a variety of operating systems and application dialects.

REVIEW BY KIM HARRIS

During the European User's Group Meeting at Utrecht, it was exciting to meet other FORTH users and learn of their activities. Attendees shared the results of projects including improved user security, an innovative file system, a microcoded FORTH interpreter,

and the extension of FORTH to new languages. This last topic illustrated a powerful use of FORTH. A translator was written in FORTH which converted a completely different computer language into FORTH source which was subsequently compiled and interpreted. The new language was designed to be readable by bank managers in France (it resembles French COBOL). But this language can be easily controlled and expanded because the FORTH translator is small, structured and itself extensible.

We also met some University of Utrecht students who have done a lot with FORTH on an Apple II. They added floating point software to FORTH and wrote some excellent high resolution graphic words. We were shown a timing benchmark of 500 floating point additions using the same floating point software but called from FORTH or microsoft BASIC.

European FORTH users are very active in experiments for improving FORTH. FIG members were pleased to meet some of them and we all look forward to sharing past, present and future developments.

STANDARDS TEAM

Twenty seven people, comprising the 1979 Standards Team, met at Avalon, CA (Catalina Island), October 14-18, 1979. The scope of their work has significantly expanded! FORTH-77 and FORTH-78 were primarily standardized glossaries for common program expression. FORTH-79 has been extended to assure (hopefully) program portability. The glossary notation has been improved, definitions of terms added, English vocalization specified for symbols, rules of usage added and address space specified.

The intent of FORTH-79 is for a common form for publication and interchange of FORTH programs. The steps before release include a final Technical Referee review and a mail vote of the Standards Team. Watch FORTH DIMENSIONS for the availability announcement.

FORTH CONVENTION

On October 20, 1979, FIG sponsored a one day convention in San Francisco. This date allowed European members attending the Computer Faire to attend. In all there were 255 attendees with 110 at a dinner for Charles Moore's "Tenth Birthday of FORTH". Technical meetings were held and Jim Berkey has tape transcripts. $16.00 for 4 tapes. Audio Village, P. O. Box 291, Bloomington, IA 47402.

FORML MEETING

The first meeting of the FORTH Modification Laboratory (FORML) was held at Imperial College, London, January 8-10, 1980. Representatives of both the European FORTH Users Group and FIG attended.

The participants identified categories of limitations and approaches toward solutions. Detailed work is to be done by individuals with progress reports to the team. Topic areas included: Virtual FORTH Machine, concurrency, lanaguage, correctness, documentation, file system, I/O and programming methodology.

Contact Kim Harris or Jon Spencer through FIG for further information.