THE FORTH APPROACH TO OPERATING SYSTEMS

Elizabeth D. RATHER and Charles H. MOORE
FORTH, Inc.
815 Manhattan Avenue, Manhattan Beach, California 90266

FORTH is a programming technique designed for interactive, on-line multi-task minicomputer applications. It features an extensible command set which leads naturally to the development of an application oriented vocabulary and operating system. FORTH combines extreme compactness with high speed performance.

1. PHILOSOPHY

The challenge offered by the development of minicomputers has met with disappointing response from the software side of the computer industry. Much minicomputer programming is still being done in assembler. High-level language development has often consisted of attempts to "squeeze" languages developed for large computers (notably FORTRAN and BASIC) into the small memories and short word lengths that characterize minis. Similarly, minicomputer operating systems have imitated the architectural principles of systems for large, batch-oriented computers (1). All of this has resulted not only in sub-optimal utilization of the mini hardware, but also in compilers and operating systems that are complicated and difficult to use, especially in the on-line environments for which most minis are intended. As a result, it has been noted that the cost of software has been exceeding the cost of the hardware in many installations (2).

It has been the goal of the authors to reverse this trend, and to take better advantage of the economics of size, cost and performance which characterize the minicomputer. This requires that the programmer's task of efficiently using the hardware be made easier.

In 1973, Koudela (2) listed several desirable features of minicomputer software which can simplify the programmer's task:

- Integration of system functions such as language processors, operating systems, utilities (e.g. for editing and debugging) and libraries into a single package through which the user can conversationally interact with the hardware;
- Programmer interaction with source language only;
- Transparent (to the user) translation of the source to machine language for high speed execution;
- Ability to modify or extend such a system easily for specific applications.

FORTH is such a system, including in a single, integrated structure an interactive high-level language, macro assembler, multiprogrammed OS and utilities, plus some other concepts such as reverse Polish Notation (3), stack organization and virtual memory for data storage. Although these techniques are not unique to FORTH, they are unusual and somewhat unfamiliar. Their use, however, means that some nice benefits, such as re-entrant routines are a natural consequence of writing in FORTH (4). As a language, FORTH supports logical structures entirely consistent with the concepts of "structured programming".

2. HISTORY

FORTH was originally developed in 1968-70 by Moore (5), and has been used by the authors and others in over a hundred minicomputer-based real-time control, data acquisition and analysis systems. In the last two years, a superset of the system has been used successfully for medium-to-large scale data base management systems.

FORTH is presently available from the authors and other sources* for the following computers: DEC PDP11, PDP8 and PDP10; DG NOVA; Varian 620; Honeywell 316/516; Prime, MODCOMP II, HP 2100 and GA SPC16. New versions may be developed relatively easily; it is anticipated that versions for the TI 980 and Interdata computers will be available during 1976. The first microprocessor implementation of FORTH for the RCA M1800 (COSMAC) processor was recently completed (6).

*Kitt Peak National Observatory, Tucson, Arizona; National Radio Astronomy Observatory, Charlottesville, Virginia; M. S. Ewing, California Institute of Technology, Pasadena, CA

3. EXECUTIVE

The FORTH executive coordinates a large number of very small modular routines, most of which are written in the high level FORTH language. These are defined as "words" in a dictionary. The dictionary is searched to obtain the "meaning" of commands typed at a terminal. Programming in FORTH thus consists of defining new words in terms of previously-defined words. These definitions may be typed at a terminal or edited (using FORTH's built-in EDITOR) into blocks of source text kept on disk. This process is superficially similar to that performed by BASIC; but the difference between FORTH and BASIC is very fundamental: BASIC is not compiled internally, but executes by parsing text instructions and performing repeated dictionary searches. This is an extremely slow and bulky technique. FORTH is compiled into compact strings of addresses of short code routines, which are executed in sequence under the control of the executive. The process of jumping to these routines adds very little overhead--typically about 20% of the actual code execution time. The amount of memory used is less than for assembler code due to the extreme modularity. An example of this effect is shown in Figure 1, which represents a typical non-trival mathematical calculation. Some comparative timings between FORTH and BASIC are given in Appendix A.

Computing $\sqrt{1 - X^2}$

```
: R1-2  DUP M* DMINUS +1. D+ SQRT ;
```

Time, µs* Execution	Time, µs* Overhead	Definition	Space, Words	Remarks
	20.5	(ENTRY)	5	Def. heading, entry/exit time
7	5.2	DUP	1	Reproduces X, on stack at entry
102	5.2	M*	1	X^2 (30-bit fraction)
10	5.2	DMINUS	1	$-X^2$
8	5.2	+1.	1	Constant providing 1.0000...(31 bits)
12	5.2	D+	1	$1.0 - X^2$
237	5.2	SQRT	1	$\sqrt{1 - X^2}$ (15 bits)
376	51.7		11	

*NOVA2, 1µs cycle time, soft MUL/DIV, all CODE definitions

Fig. 1 - Timing, overhead time and space cost for a calculation performed in FORTH. The actual definition is shown above the table. ':' begins the definition and 'R1-2' is the name of the function being defined. The remaining words are functions to be performed when R1-2 is invoked. ';' ends the definition. Memory required is 11 16-bit words; the execution time is 427.7 microseconds, of which 51.7 microseconds, or 12.1% is interpreter overhead.

FORTH routines are naturally re-entrant due to the use of the push-down stack for parameter passing. They thus may be shared amongst several users in a multiprogrammed environment each of whom has private stacks. Recursive routines are uncommon, but may be defined easily.

FORTH's version of a real time operating system is extremely small (a few hundred words) and efficient. Along with a similarly small and efficient multiprogrammer, it is capable of supporting a number of asychronous equipment control and data acquisition tasks, plus multiple interactive terminals, in a modest (8-16K word) CPU. The characteristics of this operating system may easily be modified for specific applications.

Virtually all FORTH systems are multiprogrammed. There is no particular limit to the number of tasks that may be supported concurrently. The maximum number to date for a single CPU (in this case a 32K NOVA 2) is about 35. Response time on this system rarely exceeds one second under peak loading. Tasks tend to fall into two categories: interactive terminals and terminal-less equipment handling tasks. A task is defined functionally (e.g., printer spooling or data acquisition). Some tasks coordinate the efforts of several asychronous devices related to a particular function.

All tasks in the system described above are memory-resident. This is possible because FORTH is extraordinarily compact. Further, although terminal tasks may have individual vocabularies, all tasks share a substantial general and application-oriented vocabulary on a re-entrant basis.

4. WHAT FORTH DOESN'T DO

Although FORTH does a lot of things that operating systems do, it really is not (or does not have) a "real" operating system in the traditional sense. One could compile a long list of things FORTH doesn't do:

FORTH doesn't support any other languages. (Since FORTH supports application oriented vocabularies, its users don't need different languages for different things. FORTH users can--and have--developed vocabularies for string processing, numerical integration, data base management, process control and many other areas for which specialized languages exist, and say that they find FORTH more flexible and more efficient in every case.)

FORTH doesn't treat I/O devices as "files". (This is a carryover from large mainframes and is almost always irrelevant in minis, leading to follies such as a 240-word printer driver in a main mini-computer manufacturer's operating system which used 40 words sending data and controls to the printer and 200 words doing things like issuing an error message for a "redundant OPEN").

FORTH doesn't check for disk errors (what it does do is preserve disk status for applications that can take appropriate action--which will be very different in a disk diagnostic and in a file management system).

FORTH doesn't maintain libraries of object code (since recompiling from disk takes only a few seconds*, and is done infrequently, this housekeeping chore is unnecessary).

FORTH doesn't handle conventional overlays. (Since FORTH applications tend to be half the size of comparable assembler programs and as much as 20 times smaller than FORTRAN object code, this is not necessary. FORTH does permit a terminal user to

*Compiling and loading the entire 16K application described in Appen. B takes under 30 seconds, on a PDP11/40, and is done once a day.

compile into his partition a selected subvocabulary which "overlays" a previous sub-vocabulary. This is functionally the same.*)

FORTH doesn't have a link loader (since there's no object code to link).

FORTH doesn't do priority queuing of multiprogrammed tasks (a simple round-robin algorithm paced by interrupt-driven I/O operations suffices for the vast majority of applications--and the multiprogrammer is available in source for handling the occasional situation that doesn't fit).

FORTH doesn't do parameter validation (this vital function can and should be performed much more intelligently in the application).

FORTH doesn't prevent the user from doing any of these things when they are relevant to the application.

5. ELEMENTS OF THE FORTH SYSTEM

FORTH is characterized by five major elements: dictionary, stack, interpreter, assembler and virtual memory. Although none of these is unique to FORTH, there is a synergistic effect in their interaction that produces a programming system of unexpected power and flexibility. We shall describe these elements briefly, then discuss their implementation. But first, since FORTH is basically a vocabulary, it is important to understand what constitutes a "word" in this vocabulary.

A word is any string of characters bounded by spaces. There are no special characters that cannot be included in a word, or that cannot start a word. Thus characters that represent arithmetic operators, or characters that resemble punctuation, can be words if bounded by spaces. For example, the following are words:

```
   FORTH  begin  +  ?  2   ,CODE  3.14  '
```

In general, the 128-element ASCII character set is supported.

5.1 Dictionary

The FORTH language is organized into a dictionary that occupies almost all the memory used by the program. The dictionary is a threaded list of variable length items, each of which defines a word of the vocabulary. The actual content of each definition depends on the type of word: noun, verb, etc. The dictionary is extensible, growing towards high memory. Terminal tasks may have private dictionaries, which are connected in a hierarchial tree structure.

Words are added to the dictionary by defining words, of which the most common is ':'. The execution of ':' causes a dictionary entry to be constructed for the word following. The definition of this new word in the form of addresses of previously defined words, will also be placed in the dictionary. The definition is terminated by ';'. For example,

```
Ex. 1    : PHOTOGRAPH  SHUTTER OPEN  TIME EXPOSE
                 SHUTTER CLOSE ;
```

might be a definition for PHOTOGRAPH. Such words act as verbs, performing various operations. Some other common types of definitions will be discussed below.

5.2 Stack

Two push-down stacks (LIFO lists) are maintained for each task in the system. These provide the primary communication between routines as well as an efficient mechanism for controlling logical flow. A stack normally contains items one computer word long, which may be addresses, numbers or other objects. Stacks are of indefinite size, and grow towards low memory.

The first, and most visible to the user, is the parameter stack. This contains all information being passed to FORTH operators, replacing conventional calling sequences. A good popular discussion of the uses and advantages of this stack is given by Burns and Savitt (3). A more definitive discussion is given by Knuth (7). The stack functions in a manner analogous to the stack in the Hewlett-Packard pocket calculators.

The second stack is called the return stack, as its primary function is to hold return addresses for nested definitions, although there are some supplementary uses of this stack. The use of stacks for return addresses is fairly common in modern compilers and is discussed in various texts such as Korn (8).

5.3 Interpreters

FORTH is fundamentally an interpretive system, i.e., program execution is basically controlled by data items rather than by machine code (9). It is a common assumption that interpreters are severely wasteful of CPU time, but this is avoided by FORTH in maintaining two levels of interpretation.

The first of these is the outer, or text interpreter. It works in a conventional manner, parsing text strings coming from terminals or mass storage and looking up each word in the dictionary. When a word is found in the dictionary, it is executed (unless the task is in compile mode, which is discussed below), by invoking the inner interpreter.

The inner interpreter interprets strings of addresses by executing the definition pointed to by each. The content of most dictionary definitions is addresses of previously defined words, which are to be executed by the inner interpreter, as in Example 1 above. When the inner interpreter executes PHOTOGRAPH it will execute the words SHUTTER, OPEN, TIME, EXPOSE, SHUTTER, CLOSE and ';' in sequence. This task requires no dictionary searches, for these words have been compiled. That is, when PHOTOGRAPH was defined (i.e., when the outer interpreter processed the text in

*On the same system described in Appen. B loading an analysis option takes about 2 seconds. This may be done roughly every few hours.

Example 1 following the execution of ':' which put it in compile mode) the dictionary was searched for each word in the definition and the resulting address placed into the entry for PHOTOGRAPH. The text that defines PHOTOGRAPH is not stored in memory, as is common with interpretive languages. Figure 2 shows the dictionary entry for Example 1. Another example appears in Figure 1.

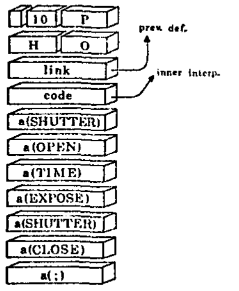

Fig. 2 - Dictionary entry for PHOTOGRAPH, as compiled from Example 1 in text.

The low-level interpreter has several important properties. First, it is fast. Indeed on some computers it executes only one instruction for each word, in addition to the code implied by the word itself. Second it interprets compact definitions. Each word used in a definition is compiled into a single memory location. Finally, the definitions are machine-independent, for the definition of one word in terms of others does not depend upon the computer that interprets the definition.

As a result, most of the words in a FORTH vocabulary will be defined by ':' and interpreted by the low-level interpreter. The high-level interpreter itself is defined in this way.

5.4 Assembler

By using the defining word CODE, the programmer can define words that will cause specified machine instructions to be executed. This type of definition is necessary to perform I/O, implement arithmetic operations, and do other machine-dependent processing.

This is an important feature of FORTH. It permits explicit computer-dependent code in manageable pieces with specific interfacing conventions. To move an application to a different computer requires re-coding only the CODE words, which will interact with the other words in a computer-independent manner. As FORTH itself only contains about five hundred instructions of code, it is also moveable with relative ease--typically 4-6 man-weeks.

The assembler is an ordinary FORTH vocabulary and therefore is implemented by the standard FORTH interpreters. Instruction mnemonics are words whose execution at assembly time causes machine instructions to be entered in the dictionary. Register designations, addressing modes, and addresses put parameters on the stack which will be used by the mnemonic operator to assemble the instruction.

The assembler is quite small. Its major cost is the dictionary space occupied by the nmemonic definitions, typically 500 bytes. The push-down stack eliminates the symbol table, a large memory requirement with conventional assemblers. It does this by effectively eliminating the need to name memory locations. Verbs find their parameters on the stack, rather than taking them out of named locations. Variables and locations, however, may be named, and such names are found in the dictionary as usual.

5.5 Virtual memory

The final key element of FORTH is its blocks--1024 byte sections of secondary memory (normally disk, although other media may be used). Two or more buffers are provided in memory into which blocks are read automatically whenever referenced. Each block has a fixed block number, which is a direct function of its physical address in secondary memory. If a block is modified in core, it will be automatically replaced on disk when its buffer must be re-used. Thus, explicit reads and writes are not required; the programmer may presume the data to be in memory whenever it is referenced.

Blocks are used to store the text that defines the vocabulary. These blocks are compiled into core when requested by a user. An editing vocabulary formats a block for display into 16 lines of 64 characters. It allows the user to modify and re-compile his source code from secondary memory.

Blocks are also used to store data. The programmer can easily combine small records into a block, or spread large records across several blocks. The fact that blocks appear to have the same physical record size, regardless of computer, makes it easy to move an application from one computer to another.

6. IMPLEMENTATION

The FORTH functions described so far are made available through an intrinsic vocabulary of about 100 defined words. This basic dictionary is resident in about 3K memory locations, on all machines on which it has been implemented so far, supplying terminal I/O, disk I/O in the form of virtual memory, interpreters, number conversion, arithmetic, assembler, compiler, multiprogrammer and text editor.

The basic vocabulary includes several commands which are, in fact, compiler directives and which give the programmer the necessary tools to control logical flow. The DO...LOOP construction resembles the FORTRAN DO. IF...ELSE...THEN is the basic conditional logic as in ALGOL (except for a "Polish" order of clauses). A set of condition evaluation commands operate on the stack to provide a truth value as a parameter for IF. BEGIN...END gives a loop which will continue until the parameter for END (which is computed inside the loop) is TRUE. Analagous structures are also available in the assembler. There is no GO TO construction: the discipline of "structured programming" is a natural consequence of FORTH programming. To the basic vocabulary will be added additional words required by the application (such as extended mathematics and an appropriate level.of file management) and finally the application itself.

The basic FORTH program is itself written in FORTH, which makes it easy to generate FORTH for a new computer, given a computer on which FORTH is presently available.

A diagram showing memory utilization in a typical application is shown in Figure 3. The shaded area at the bottom is the only precompiled portion of the program. As much of the system as possible is kept in source form, to facilitate changes and additions. This costs little, since to re-compile the entire system and application into memory takes only a few seconds. Re-compiling is rarely necessary on an operational system.

You will see that each terminal task has a partition which contains its stacks and into which may be compiled a selected vocabulary to do some particular kind of processing which is a subset of the vocabulary, but is not available to users at other terminals. The data acquisition task MONITOR has a much smaller area, with only enough space for his stacks. The routines it executes are located in the common area.

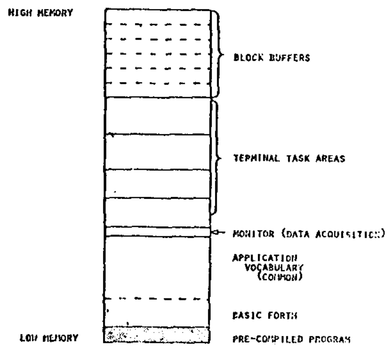

Fig. 3 - Memory allocation in a typical system.

7. AVAILABLE SYSTEM FUNCTIONS

Most of the words used by FORTH are available to the programmer to perform system-related functions. Some of these are summarized briefly here:

Usage	Description
n EXPECT	Accepts up to n characters from a terminal into the user's input buffer. The string may be terminated early by a RETURN character, where $0 \leq n \leq 80$.
PAD n TYPE	Sends to the user's terminal a string n characters long located at PAD (an address), where $0 \leq n \leq 128$.
b BLOCK	Returns to the user the address in memory of the first word of disk block b, having read it if necessary (i.e. if it wasn't already in memory), where $0 \leq b \leq$ capacity of mass memory in 1024-byte blocks.

Usage	Description
m WORD	Parses the input message buffer using the constant m (an ASCII character code) as delimeter, moving the resulting string to a standard location, along with its length.
(word)	Searches the dictionary for (word) and returns the location of its definition, where (word) is a character string containing up to 64 non-blank characters.
VOCABULARY (name)	Defines a chain of definitions, which will be linked to the current chain named (name). (name) is an ASCII character string with no embedded blanks. Subsequent use of (name) will specify this chain as the one in which searches begin.
(name) DEFINITIONS	Specifies that future dictionary entries will be linked to the chain named (name) defined by VOCABULARY above.
QUIT	Stop execution for this user and return to the terminal for control.
n QUESTION	Aborts this user, emptying his stacks and issuing error message number n.
r s MEMORY (task)	Defines a terminal-less task named (task) with s words of parameter stack and r words of return stack.
(task) ACTIVATE RUN n STOP	ACTIVATE starts the task named (task), defined using MEMORY above. RUN performs some previously defined function, n STOP stops him, leaving his status set to the integer value n.
(name) GET	Attaches facility named (name) to this user if it is available. Waits if it is in use.
(name) RELEASE	Releases the facility named (name) NOTE: The GET...RELEASE mechanism may be used to control access to both logical and physical facilities, ranging from disk to a non-re-entrant software function.
(task) PROMPT	Forces the task named (task) to abort whatever it is doing.

8. DEVICE DRIVERS

FORTH systems support all hardware devices attached to a particular computer. However, in recognition of the tendency of mini systems to add or change peripherals, all device drivers are supplied in source form, to be loaded optionally. For example, it is common to find a tape drive used only for occasional backups. In such a case, a user would load the tape driver as part of the backup utility vocabulary.

To see a non-trival example of FORTH in use, consider the tape driver in Figure 4, coded for the NOVA computer. The two blocks shown compile into about 150 words of memory. They are designed to be loaded into the user partition of someone wishing the use of the tape driver.

```
64
     0 ( TAPE I/Ø )        TAPE GET
     1 ØCTAL 0 VARIABLE STATUS   U @ VARIABLE TAPE
     2 CØDE COMMAND  0 S) 0 LDA   22 DØA 1B  WAIT
     3 CØDE TRANSFER   0 S) 0 LDA  0 0 NEG   22 DØC
     4    1 S)  0 LDA   22 DØB  2PØP
     5 BEGIN   TAPE ) 0 STA   22 DIA OD   STATUS 0 STA  22 INTERRUPT
     6
     7 CODE REWIND   TWO 6 + 0 LDA   22 DØA 1B   NEXT
     8 : READ   TRANSFER  0 COMMAND ;
     9 : WRITE   TRANSFER  50 COMMAND ;
    10 : BACKSPACE   0 1 TRANSFER  40 COMMAND ;
    11 : SKIP   0 0 TRANSFER  30 COMMAND ;
    12
    13 : PARITY   STATUS @ 2040 AND ;
    14 : EØF  STATUS @ 400 AND ;
    15 DECIMAL

65.
     0 ( TAPE ERRØR RECØVERY )
     1 0 VARIABLE ERRØRS
     2 : (READ)   2DUP READ  PARITY IF
     3    1 ERRØRS +!  BACKSPACE  2DUP READ  PARITY IF
     4         BACKSPACE  2DUP READ  THEN THEN 2DRØP ;
     5
     6 ØCTAL : ERASE   70 COMMAND ;    DECIMAL
     7 : (WRITE)  2DUP WRITE  PARITY IF  BEGIN
     8        1 ERRØRS +!  BACKSPACE ERASE  2DUP WRITE  PARITY NOT END
     9   THEN 2DRØP ;
    10
    11
    12
    13
    14
    15
```

Fig. 4 - Example showing magnetic tape I/O. Lines 2-5 of block 64 coded for the NOVA computer.

The key words defined are as follows (Figure 4)

STATUS	variable containing tape status as of the last operation (set by the interrupt handler).
TAPE	variable (defined previously) containing the address of the user of the tape. The phrase TAPE GET, executed at compile time checks that no one else is using the drive, and sets the current user's address in TAPE.
COMMAND	issues a command to the tape controller and waits for an interrupt indicating completion. The command code is on the stack. COMMAND ends with WAIT, a macro which assembles a return to the multiprogrammer with this task set inactive pending an interrupt.
TRANSFER	sets up the parameters for a DMA transfer to the tape. It requires a starting address and count on the stack.
REWIND	issues a "rewind" command. Unlike all other tape commands, it does not get an interrupt from the controller on completion. Thus, this code ends with NEXT (inner interpreter return) instead of WAIT (multiprogrammer return).
READ and WRITE	each use TRANSFER (with parameters on the stack) and then issue an appropriate COMMAND).
SKIP and BACKSPACE	are similar to READ and WRITE except that dummy parameters required by the controller are supplied to TRANSFER.
PARITY and EOF	test STATUS for certain condition bits, leaving a truth value on the stack.
ERRORS	is a variable which "remembers" how many tape errors have occurred. It may be queried and reset elsewhere.
(READ)	performs a READ, with source address and count on the stack. It will try up to three times for an error-free transfer, and finally will accept what it gets. Only one error is tallied.
(WRITE)	as in (READ) performs a WRITE with error checking, plus tape erasure if recovery is necessary. (WRITE) will try indefinitely to write correctly (hoping eventually to come to good tape after leaving a long erased area).

Line 5 contains the interrupt handler for the tape. It is not a normal FORTH definition, as it has no name and no heading. Instead, it is merely a string of machine instructions assembled into the dictionary. At assembly time, BEGIN pushes the address of the beginning of the string onto the stack; INTERRUPT takes this address and enters it into a table of interrupt vectors, in the place appropriate for device 22; it then assembles a jump to the interrupt return code which is standard for the NOVA.

FORTH normally runs with interrupts enabled. When an interrupt occurs, FORTH's NOVA interrupt handler saves 2 registers and executes a vectored jump to the code whose address is in the appropriate table entry for the interrupting device. This interrupt handler returns to instructions which restore registers and resume processing.

Although this mechanism is heavily machine dependent, every effort is made to standardize the protocol, so that all FORTH interrupt handlers appear very similar.

The main principle is that all logic is performed by high-level routines such as (READ), rather than at interrupt time. Being high level, (READ) may be easily modified to handle errors differently; this would be much more difficult if error handling were embedded in a code driver. Moreover, having the very simple routine READ available means it will be easy to use READ in a diagnostic routine which would probably want to handle errors quite differently from (READ).

9. SUMMARY: FORTH PROGRAMMING

The task of programming an application consists of identifying the functions to be performed, specifying the block or blocks to be used, defining a preliminary top level user vocabulary, and then defining all the words used to implement the user words. These will have to be loaded in reverse order as well.

Testing consists of putting reasonable parameters on the stack and typing the word to be tested. This applies both to CODE definitions and ':' definitions. Any memory location may be examined ('?') or a region may be dumped at any time, in any number base. These simple but powerful aids, plus extreme modularity (most routines are about 20 instructions long) make FORTH applications unusually easy to debug interactively.

Over the past several years, FORTH has been used in a wide variety of application areas: on-line data acquisition, analysis, interactive graphics, image processing and data base management. The goal in every case has been to preserve in the language the ability to solve problems at hand easily and efficiently, without imposing the penalty in efficiency of trying to do everything in one system.

The success of FORTH in retaining its small size and simple structure has paid an additional dividend. Basic FORTH contains only about 500 explicit machine instructions. To code FORTH for a new computer means replacing these instructions with their functional equivalents. Thus, it is possible to prepare FORTH for a new computer in several man weeks. Present work on microprocessor systems (6) confirms the universality of the FORTH technique.

REFERENCES

(1) D. L. Mills, Executive systems and software development for minicomputers, Proc. IEEE, vol. 61, November, 1973, 1556-1562.

(2) J. Koudela, Jr., The past, present and future of minicomputers, Proc. IEEE, vol. 61, November, 1973, pp. 1526-1534.

(3) R. Burns and D. Savitt, Microprogramming and stack architecture ease the minicomputer programmer's burden, Electronics, vol. 46, 15 February 1973.

(4) P. Stein, The FORTH dimension: Mini language has many faces, Computer Decisions, November, 1975, pp. 10.

(5) C. H. Moore, FORTH: A new way to program a minicomputer, Astron. Astrophys Suppl. 15 1974, pp. 497-511.

(6) E. D. Rather and C. H. Moore, FORTH high-level programming technique on microprocessors, paper presented at Electro 76 Professional Program, Boston, MA, May 11-14, 1976.

(7) D. E. Knuth, The art of computer programming, vol. I. Reading, MA: Addison-Wesley, 1968.

(8) G. A. Korn, Minicomputers for Scientists and Engineers. New York: McGraw-Hill, 1973.

(9) M. S. Ewing, The Caltech FORTH manual, Owens Valley Observatory, California Institute of Technology, Int. Rep. 1974.

(10) C. H. Moore and E. D. Rather, The FORTH program for spectral line observing, Proc. IEEE, vol. 61, September, 1973, pp. 1346-1349.

(11) E. R. Fisher, High level languages in mini-computer automation, paper presented at Electro 76 Professional Program, Boston MA, May 11-14, 1976.

APPENDIX A

```
  5  HWAM = 14004Q
 10  REM "THIS IS A SIMPLE BENCHMARK PROGRAM.
     IT"
 15  REM "MULTIPLIES, DIVIDES, ADDS AND SUB-
     TRACTS.
 20  REM
 30  INPUT A
 40  INPUT B
 50  LET C = A + B
 60  LET A = A + 1
 70  LET E = B/C
 80  LET F = A * E
 90  LET C = C - F
100  IF A = 1001, THEN 200
110  GO TO 50
200  PRINT "THE LOOP IS DONE AT"
210  END
```

LLL BASIC benchmark program (11). The compiled version for the Intel 8080 microprocessor occupied 896 bytes memory.

```
: GO   1001  SWAP DO  DUP I +  2DUP I 1+
         SWAP */ - DROP LOOP ;
```

FORTH equivalent of the LLL benchmark.
The user types
      b a GO

where 'a' and 'b' correspond to the entry of A and B in the BASIC version. GO compiles into 42 bytes on most computers. The COSMAC version required 54 bytes.

Timings were obtained in all cases by running the loop a sufficient number of times to get an accurate measurement and dividing by that count to get the timing for one pass. The results are given in Table A.1.

INTERPRETER	PROCESSOR	MILLISECONDS/LOOP	MUL/DIV
DG MULTI-USER BASIC (ONE USER DURING BENCHMARK)	NOVA 840	4.5	HARDWARE
FORTH	NOVA 2/10	.395	SOFTWARE
DEC BASIC PLUS	PDP11/45	3.2	HARDWARE
FORTH	PDP11/40	.235	HARDWARE
INTEL BASIC	INTEL'S 8080	75.	UNKNOWN
LLL COMPILED BASIC	8080	22.	UNKNOWN
FORTH	RCA CDP1802 (COSMAC)	6.1	SOFTWARE

Table A.1 Comparison of FORTH with BASIC on some similar computers. All BASIC timings from E. R. Fisher, Lawrence Livermore Laboratory (11).

Programming of the FORTH loop was done by three independent people with varying degrees of expertise in FORTH: an expert, an experienced programmer/engineer, and a novice with less than a month's experience in FORTH. Minor differences appeared in the resulting definitions, with less than 10% effect on space or time. The clock time required for each person to code, test and time the loops was 6 minutes, 11 minutes, and 15 minutes, respectively.

Comparing these figures with those given by Fisher using other techniques on microprocessors, yields:

Language	Processor	Man-Hours	Bytes
PLM	8080	16	1172
LLL BASIC	8080	1	896
Assembly	8080	32	98
FORTH	RCA 1802	.4	56

The RCA 1802 microprocessor has timing features comparable to the Intel 8080. Preparation of FORTH for the 8080 is underway.

APPENDIX B

A PROJECT EXAMPLE

PROBLEM: Radio Telescope System (10),
- --Telescope pointing
- --Data acquisition and recording
- --On-line interactive graphics analysis
- --3 observing modes

TYPICAL SYSTEM ESTIMATES:
- --6 man-years programming
- --$120K hardware costs
- --64K words memory
- --2 calendar years

RESULTS USING FORTH:
- --24 Man-weeks programming
- --$50K hardware costs
- --16K words memory
- --12 calendar weeks