A VAX Implementation of the FORTH-79 Standard

William L. Sebok

Princeton University
Department of Astrophysical Sciences
Princeton, NJ 08544

Abstract

A public domain implementation of Forth has been written for the VAX[1] super-mini computer that runs under 4.1 and 4.2 BSD UNIX[2]. It has been running now for over two years at the Astrophysics Department at Princeton University and is used for image processing. It follows the 79 Standard except: 1) entries on the parameter stack are 32 bits wide, 2) addresses are 32 bits (rather than the demanded 16 bits) wide, and 3) certain escape sequences beginning with a backslash are recognized in the printing word.

Some extensions to the 79 Standard are: 1) a character string stack, with a full set of string operators; this also makes manipulation of UNIX file names infinitely easier, 2) a floating point stack, and 3) a set of UNIX interface words.

Colon definitions are compiled as a series of *bsb, bsbw,* or *jsb* instructions rather than as a list of pointers. The ICODE operator can be used instead of the CODE operator for short definitions. When a word defined by the ICODE operator is compiled, its code is placed in-line rather than referenced. Number references are compiled as the shortest of the many possible instructions to push that number onto the stack.

Introduction

An implementation of the Forth-79 Standard has been written at the Princeton Astrophysics Department to run on a VAX super-minicomputer. This Forth implementation is intended for use in image processing and pattern recognition. Image processing involves the manipulation of large arrays of numbers. Thus, the emphasis has been on speed. The kernel is written in VAX assembler, as the calling conventions are rather different than those of other languages. The rest of the basic Forth definitions are written in Forth. It was felt that the speed of execution would be unacceptable if it were written in a higher level language such as "C".

Other VAX Forth implementations have been written by Wayne Hammond at the Jet Propulsion Laboratory and Rich Stevens and others at Kitt Peak National Observatory [1,2]. This implementation was written independently of either of them, although it has been influenced by both.

[1] VAX is a trademark of Digital Equipment.
[2] UNIX is a trademark of Bell Laboratories.

History

The ancestor of this Forth implementation was a PDP-11 stand-alone Forth that ran on a PDP-11/34 at the Caltech Astronomy department. Some of the features of the compiler that existed at Caltech in 1981 were:

1. Use of the PDP-11/34 memory management to place code (not just data) partitions in high memory using a common 16K byte partition for communication between processes.
2. Locking and unlocking of block buffers for better asynchronous sharing of the block buffer pool.

This compiler was used for image processing and pattern recognition [3]. It had a fairly extensive library of image processing routines. The most important application of this sytem had three linked processes that would simultaneously:

1. Control a photograph digitizer and remove the signature of the light detector from the digitized output.
2. Detect the objects in the digitized output.
3. Classify the objects as stars or galaxies, etc. and measure their properties.

This application used 136 Kb of memory and could handle 3000 measurements (pixels)/second. It was used to generate a catalog of 3 million objects from the processing of 4 plates [4,5]. Each plate contains the equivalent of a billion pixels.

The primary concern was code efficiency and speed. Thus, techniques were sought that could be implemented efficiently, and there was an emphasis on CODE words in the innermost loops that had to handle all of the pixels in the image.

At Princeton this Forth system was implemented under UNIX, initially version 6, but later System III. At the same time, an effort was undertaken to bring this Forth into compliance with the Forth-79 Standard.

In 1982 the department PDP-11/60 was replaced by a VAX-750 running BSD 4.1 UNIX. Efforts to run the PDP-11 Forth under BSD 4.1's primitive PDP-11 compatibility mode were not successful, as compatible mode support in BSD 4.1 is poor. At that point the effort was undertaken to write a Forth that would run full VAX native mode code. The kernel and all machine dependent words were rewritten. This effort and its debugging took place in 1982, and slow evolution has continued since then.

Machine Compatibility Issues

It was decided that the declaration in Forth-79 that pointers are 16 bits wide was unacceptable. Perhaps the major reason for the migration from a PDP 11/60 to a VAX 750 was to remove the 64 Kbyte/process addressing restriction imposed by the PDP 11's use of 16 bit wide pointers. Retaining this restriction, when the major reason for the hardware switch was the removal of that restriction, was intolerable.

Forth faces a major difficulty when it attempts to support more than one data type. Different data types often have different natural lengths on machines with different architectures. This creates a problem when an attempt is made to manipulate these quantities on a stack. If the item is not on the top of the stack, then its relative length must be known. Three ways of handling this problem have been tried:

1. Define a "standard length" for an item. This penalizes architectures for which the natural size for the item is not the standard length. In particular it penalizes architectures for which the natural length is greater than the "standard" length. An example of this problem is the need for pointers to a 32-bit-wide address space when the Forth-79 Standard defines pointers to be 16 bits wide.

2 . Define a separate stack for each data type. This allows each data type to be its own natural length. It is impractical to carry this approach through completely, giving every possible data type its own stack. It results in fragmentation of memory to support the different stacks. Still, it may be useful to give a major data type or class of data types its own stack. The Forth implementation discussed here has a stack for floating point and string items, as well as the usual parameter and return stacks.
3 . Let the length of an item be its natural length, but restrict the number of stack manipulations allowed on them. Providing a complete set of operations for every data type is awkward. It is even worse to have to provide versions of every operator of two or more operands for each possible combination of operand types. An example of this minimal set of stack manipulation operators is:
   a) Conversion operators (where practical) to convert one item type to another.
   b) Operators to swap any two items of whatever type on the stack.
   c) Operators to move an item to/from the return stack.
   d) Operators to move an item between the stack and memory.

An item of arbitrary type covered arbitrarily deeply with other items of different arbitrary types can be brought to the top of the stack in a machine independent manner by a combination of the inter-item swap words and the words to move the items to and from the return stack.

In this implementation, a combination of these methods is used. It had been decided that pointers would be 32 bits in width. Since some of the most common code is that which manipulates pointers and integers, it was decided that it would be simplest also to double the size of integers and make the standard stack entry 32 bits wide.

A set of address arithmetic operators is provided. If one limits oneself to using only these operators for address arithmetic, then one is able to compute the address of the nth element of an integer array in a machine independent fashion.

Floating point numbers have their own stack. It was first considered to extend the philosophy of doubling the sizes of everything to floating point numbers, thus making all floating point numbers double precision. Since there is intent of using this Forth implementation for floating point "number crunching", this approach was rejected.

Character strings also have their own stack. While being useful in their own right, characters' strings on the string stack have also been very useful for communication with the UNIX operating system.

Addressing

The following words greatly eased address arithmetic. On the VAX implementation the word length is 4 bytes, while on the PDP 11 implementation the word length was 2 bytes. The transition to the VAX was made much easier by limiting address arithmetic to these words.

The notation in the following definition is that the items to the left of the triple dash are the contents of the stack before the operation, while the items to the right of the triple dash are the items left on the stack after the operation. The top of the stack is to the right.

`BYTE` *n1 -- n2*
Convert a word offset to a byte offset, i.e., multiply by 2 or 4, whichever is the word length in bytes.

`WRD` *n1 -- n2*
Convert a byte offset to a word offset, i.e., divide by the word length in bytes.

`++` *a1 -- a2*
Increment address *a1* by one word-length, i.e., add the word length in bytes to address *a1*.

`--` *a1 -- a2*
Decrement address *a1* by one word-length, i.e., subtract the word length in bytes from address *a1*.

`A+` *a1n -- a2*
Add word offset *n* to address *a1*, i.e., multiply n by the word length in bytes and add to address *a1*.

`+A` *na1 -- a2*
Add word offset *n* to address *a1*, i.e., multiply *n* by the word length in bytes and add to address *a1*. Like `A+` except arguments are swapped.

`A-` *a1n -- a2*
Subtract word offset *n* from address *a1*, i.e., multiply *n* by the word length in bytes and subtract from address *a1*.

Floating Point Stack

The floating point stack allows floating point values to have their own length, independent of the length of integers and pointers. Actual use of the floating point stack has shown that it greatly simplifies words that do floating point computations, especially on arrays. Floating point computations are often split into the computations themselves and address or index manipulation. The presence of a floating point stack allows these components to proceed less independently, with much less stack manipulation required.

The floating point stack is fully incorporated into the Forth kernel. Thus, the floating point stack is cleared on abort and the outer interpreter checks for floating point stack overflows and underflows after the interpretation of every word.

String Stack

A set of words is provided for manipulation of the string stack, derived from the University of Rochester string handling package [6].

The first byte of a string on the string stack contains the number of bytes in the string (not including the length byte itself). This implies that the length of strings is less than or equal to 255 bytes. Zero length strings are allowed. In addition, there is a null byte after the end of the string which is not included in the length count. This simplifies communication with the UNIX operating system, which requires string-valued quantities, like file names, to be delimited by a null character.

The string stack is fully incorporated into the Forth kernel. Thus, the string stack is cleared on abort and the outer interpreter checks for string stack overflow and underflow after interpretation of every word. Low level error handling routines use the string stack for passing messages.

Several special "escape sequences" are recognized in the specification of a string to allow control characters in strings. Escape sequences include a backslash and one or more following characters, which together are interpreted as a single, usually printable character. These "escape sequences" are the same as those used in the character strings of the "C" language, with some extensions.

They are:
\ b = backspace
\ e = escape char (octal 033)
\ f = form feed
\ n = new line (line-feed)
\ r = carriage return
\ t = tab
\ ? = rubout character (octal 0177)
\ \ = backslash character

\ nnn = nnn is up to 3 digit octal number
\ X = X is a capital letter; value of X with most significant 3 bits removed (i.e., \ b = "control C").

All other characters following a backslash yield an undefined result.

Memory Allocation

The Forth kernel allocates virtual memory from the VAX's large address space in a manner that is transparent to the user's program. Memory on a UNIX system is laid out in a manner similar to the way memory is laid out in Forth. Starting at low addresses, there is the text segment, a read-only area consisting mainly of executable instructions. Above this is the data area, which is both readable and writable. Then there is a large gap in address space above which the stack area grows down from the highest addresses. The UNIX kernel automatically adds more memory to the stack area as it senses its growth. The data area can also be expanded by the UNIX *brk( )* system call, which changes the location of the highest legal address in the data area. This allows one to allocate or release virtual memory in the data area.

This memory layout matches well with the Forth memory layout, in which a dictionary grows upward from low addresses and stacks grow downward from high addresses. This makes it easy for the Forth kernel to manage memory.

Memory allocation is checked in two places, by `ALLOT` and by a consistency check in the outer interpreter done after the execution of every word. A buffer zone size (currently 256 bytes) is added to the dictionary pointer and the result rounded upward to the next memory page. This is then compared to the previous value of that quantity. If a difference is detected, then a *brk( )* system call takes place, allocating new virtual memory from the operating system. As in non-virtual memory systems, small amounts of memory can be allocated by incrementing the dictionary pointer. The test in the outer interpreter ensures that the extra virtual memory will eventually be allocated. The test in `ALLOT` ensures that large memory allocations will take place immediately. The 256 byte buffer zone mentioned above is intended for words that use the memory at the address of the dictionary pointer for scratch space.

The intent of this memory management strategy is to allow easy use of large amounts of memory for applications that might need it, like image processing, while not loading the resources of the operating system with large fixed partitions for applications that do not need them. In practice the amount of memory is set by a system quota, typically 6 Mb on a 4.2 BSD UNIX system, which enables users to type `5000000 ALLOT`, for instance.

File References

The input stream for Forth can either be a terminal or a disk file. Both files containing traditional 1024 byte Forth screens and ordinary UNIX text files are supported. One of the design goals was to allow loads of any combination of these files to nest to any level (consistent with the depth of the return stack).

A "block mapping table" is used to access "screen" format files. The use of this table is best shown by example. Suppose a file /usr/src/forth/cos contains 3 Forth screens. After executing the command:

```
100 INSTALL /usr/src/forth/cos
```

blocks 100, 101, and 102 will be defined and point to the three blocks in the file. When one is finished with this file, one can type:

```
100 REMOVE
```

to remove this file from the block mapping table. Then, attempts to access blocks 100, 101, 102 will

again abort with an "Undefined Block Number!!!" message. Several mappings can be active at the same time, allowing one to move text between files.

One can load this file with:

```
LOADF /usr/src/forth/cos
```

which allocates a free range of block numbers for the file, LOADs it, and then removes it. One can also load it with:

```
FLOAD /usr/src/forth/cos
```

which uses a different mechanism (intended for use with text files) that does not involve block numbers.

This implementation also understands the existence of a standard directory for file names given to the above commands. This directory name is contained in the string returned on the string stack by the word FDIR. If this directory name were /usr/src/forth/, then it would have been sufficient to specify the last three commands given above as:

```
100 INSTALL cos
```

```
LOADF cos
```

and

```
FLOAD cos
```

A search is first made to the current directory before checking for the existence of the file in the "standard" directory. Prefixing the file name with a < will cause only the standard directory to be searched. Thus:

```
100 INSTALL <cos
LOADF <cos
FLOAD <cos
```

guarantees that only /usr/src/forth/cos will be referenced.

This mechanism has recently been extended with one that uses a user-settable directory search path, so that one can designate a directory as a standard place for one's own Forth definitions.

To Execute a Word at the End of a Line

A mechanism is provided to cause an action to occur before input is requested from the terminal. Thus ' word !SIGNAL will cause "word" to be executed (with no parameters) when input is next requested from the terminal. 0 !SIGNAL will cancel that request.

This facility is used mainly to flush buffers on interactive graphics. The cost of unbuffered graphics in an operating system environment is too severe to be tolerated. It is likely that this facility will be replaced in the future by a mechanism that maintains a queue of such requests.

Internal Assembler

Like most other Forth implementations, there is an internal assembler for the creation of CODE definitions. The assembler is especially important here as there is heavy use of the assembler words to aid in the compilation of higher level definitions. Writing an assembler for the VAX was easy because of the near orthogonality of instructions and their operands. In other words, in an instruction with n operands, each operand is constructed in the same way. Thus, there was a simple implementation of assembler words; each assembler word has in its parameter field a null-terminated byte string containing an op-code and a string of descriptor bytes, one per operand.

The VAX has a wealth of registers that can be used in Forth. The register assignments in this implementation are shown in Table 1. Registers 0-6, 12, 13 are unused and can be considered

scratch. One cannot assume that they will remain intact between words. The letter names are defined in the ASSEMBLER vocabulary.

Table 1

Register	Name	Use
14	R	Return (also system) stack pointer
11	S	Parameter (integer) stack pointer
10	F	Floating point stack pointer
9	C	String stack pointer
8	H	dictionary pointer
7	U	User area pointer

Having the dictionary pointer as a register simplifies compilation and simplifies the words that use the beginning of free memory (i.e., the memory pointed to by H).

The user area is an area of memory that contains the process state of the Forth compiler. Addressing it with a register is a relic of this Forth's stand-alone ancestry, where multiple processes sharing the same memory were supported. User area register addressing is retained here because eventually, when better operating system support of asynchronous I/O makes it worthwhile, co-process facilities might be re-introduced.

Header

The layout of the header of a word is shown in Table 2. Offsets are given from the parameter address (i.e. the address returned by '). Only two of the header fields are mandatory, thus the minimum header size is 6 bytes.

Table 2

name	optional?	offset	bytes	
in-line length	yes	-17 or -13	1	
long link	yes	-16	4	
short link	no	-12	2	
name field	no	-10	4	
jsb @#address	yes	-6	6	code address
		0		parameter address

flag bits in name field

name	value	meaning
IMMEDIATE	1	execute immediately
IN_LINE	2	compile code in-line
SMUDGE	4	make entry unavailable

Packing Format

bit number	3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
fields	5 5 5 5 5 5 4 4 4 4 4 4 3 3 3 3 3 3 2 2 2 2 2 2 1 1 L L L H N 1

The header field has 29 bits of header and 3 flag bits. The IMMEDIATE bit causes the word to be executed rather than compiled, even if Forth is in compile mode. The IN-LINE bit causes the word to be compiled by copying the number of bytes specified by its length field, starting at its code address. The SMUDGE bit is included in the comparison of the name field with its desired value and if set, guarantees that the comparison will fail. It is used to mark the header of a word invalid while under construction.

The bits expressed in the name field of a word include the first five characters (shown as digits 1-5), plus the number of characters in the name of the word. A 6 bit ASCII subset is used for the characters, folding upper and lower case together and ignoring control characters. Only the low order 3 bits of the character count are used. However, since names 1-4 characters in length contain padding blanks, lengths of names up to 12 characters in length are distinguishable. As a 16-way multiple thread is used for the dictionary chain, 4 of the $5 * 6 + 3 = 33$ bits that otherwise would be needed are redundant, enabling the name field to fit in 29 bits.

Before the name field is the short link field. The short link field is interpreted as an unsigned 16 bit number that is the negative of the offset to the name field of the previous word in this thread of the dictionary chain. If the short link contains a 0, then it is assumed that the short link would not reach and a long link field is used. The long link field, when used, contains the absolute address of the name field of the previous word in the dictionary chain.

If the IN-LINE bit is set in the name field, then there exists a 1 byte length field, which contains the number of bytes to copy from the code address of the word when it is compiled.

Following the name field is the code address. For CODE words and words defined by colon definitions, the rest of the word begins here. For CONSTANTs, VARIABLEs and any other operator defined by a word containing ;CODE or DOES> the code field contains a *jsb* to the absolute address of code that will handle the processing of the word. (Using the worst case *jsb* to an absolute address rather than the shortest VAX instruction that will reach is necessary so that the offset between the code address and parameter address is fixed.) In this code the parameter field of the word is accessed by the contents of the top of the return stack (which must be popped before a return can be taken using the return address beneath). The implementation for DOES> allows ' (tick) to access the parameter field of a word defined by a word that uses the DOES> construction.

Token Parsing and Compilation

Word execution and compilation takes place in a fairly conventional manner. WORD obtains a token from the input stream. WORD here departs slightly from the standard in that when given 32 (space) as a delimiter both spaces and tabs are treated as delimiters. An attempt is made to look up the token in the Forth dictionary. If that attempt succeeds, the word corresponding to the token is compiled or executed, depending on whether the interpreter is in compile or execute state. If that attempt fails, an attempt is made to treat the token as some kind of constant, to be pushed onto the appropriate stack. Which kind of constant and which stack depends on the form of the token. If the interpreter is in execute mode, the constant is left on the appropriate stack. If the interpreter is in compile mode, code is compiled that when later executed will result in the pushing of the constant onto the appropriate stack. If the token is not in the form of any legal constant, an abort is taken, resulting in the token being printed on the terminal followed by a space and a question mark.

Executing or Compiling a Word

Execution of a word is done by a jump to subroutine *(jsb)* to the code address of the word. Execution starts at the code address with the return address on the return stack. The word returns at the end of execution by executing a return from subroutine *(rsb)* instruction. There is no instruction pointer register except, of course, the program counter itself.

What happens when a word is compiled depends on the setting of the IN-LINE bit in the header of the word. If this bit is not set, a branch or jump to subroutine is compiled whose

destination is the code address of the word. One of three similar instructions can be compiled depending on the distance to the destination (see Table 3).

Since the jump is to a previously compiled word, the relevant offsets will all be negative. It is expected that *bsbw* will be the instruction used in most cases. In that case this "compiled" implementation of Forth takes less memory (3 bytes per reference) than a 32 bit threaded code implementation (4 bytes per reference).

A different action is taken when a word is compiled that has the `IN-LINE` bit set in its header. The word behaves like a macro whose code is copied into the definition being compiled. The copying starts at the code field of the referenced word and continues for the number of bytes specified by a special "length" field in its header.

Colon definition words, words defined by the definitions of the form:

```
: name ... ;
```

are referenced by *bsbb/bsbw/jsb* instructions. Likewise, so are Code words, words whose definitions are of the form:

```
CODE name ... END-CODE
```

Table 3

instruction	distance to destination	bytes occupied
bsbb	-128 to 127	2
bsbw	-32768 to 32767	3
jsb (absolute mode)	< -32768 and > 32767	6

However, there also exists a new type of Code word defined by `ICODE`, which uses definitions of the form:

```
ICODE name ... END-CODE
```

`ICODE` defines a Code word with the `IN-LINE` bit set in the header.

For all this to work correctly `END-CODE` must do some extra actions. `END-CODE`

1. Compiles an *rsb* (return from subroutine) instruction.
2. If the `IN-LINE` bit is set in the header, completes the Length field in the header. This length does not include the implicit *rsb* instruction.

Having `END-CODE` implicitly compile as *rsb* (i.e. the equivalent of a `NEXT`) is necessary to make `ICODE` work both as a macro when in compile mode and directly when executed from the outer interpreter (and also to complete the symmetry between `CODE` and `ICODE` definitions). When the definition is executed, a concluding *rsb* is needed for the return. When the word is used as a macro, the return is to the code following the macro.

There are some special requirements that the code inside an `ICODE` definition must follow. Words must not directly address items on the return stack if they are not to be "compile-only." This is because items are a different depth on the return stack inside a macro than they are when the word is called from the outer interpreter (when the return address is on the top of the stack). Ordinarily this has not been a problem. The code must be position independent. This implies that references addresses of other words be absolute rather than relative. This is the reason why no in-line equivalent of a colon definition has been provided. References to execute other words are often compiled as *bsbb* or *bsbw* instructions, which being relative instructions, violate this rule. Some time in the future one could consider a compile-time flag which would be enabled by `ICODE` or the putative in-line

colon word and be reset by END-CODE or ; (semi-colon). This flag would cause all references to addresses outside the definition to be absolute.

Ordinarily, words defined as ICODE definitions should be short, no more than one or two instructions long. This was another reason that an in-line colon definition word was not provided. It was not considered necessary. Examples of words that have been defined by ICODE definitions are DUP, OVER, DROP, !, @, and +.

Executing or Compiling a Constant

The rules in this implementation for interpreting a contant are:

1. A token that begins with a backslash is considered a string constant to be placed on the string stack (without the initial backslash). Before the token is pushed on the string stack, special character sequences beginning with backslash ("escape sequences") are converted to their equivalent control characters. The initial backslash is not considered to begin an escape sequence.
2. A set of digits is interpreted as a single length (32 bits) integer to be placed on the parameter stack. A digit can include letters consistent with the current BASE, under the usual convention that A = 10 (Decimal), B = 11, etc.
3. A set of digits, including letters consistent with the current BASE, containing a , (comma) is considered a double-length number (64 bits).
4. A set of digits containing a . (decimal point), an "e", or "E", with one or two following digits is considered a floating point number to be pushed onto the floating point stack. Since the test for a digit is made before the test for "e/E" as a floating point indicator, a token containing "e" or "E" will be interpreted as an integer if the BASE is HEX, but as a floating point number if the BASE is DECIMAL. The digits after the "E/e" are considered the exponent.
5. A set of digits containing a "d", or a "D", with one or two following digits is considered a double precision floating point number to be pushed onto the floating point stack. Since the test for a digit is made before the test for "e/E" as a floating point indicator, a token containing "d" or "D" will be interpreted as an integer if the BASE is HEX, but as a floating point number if the BASE is DECIMAL. The digits after the "E/e" are considered the exponent.

An attempt is made to compile a constant as the smallest instruction that will push that value on the appropriate stack. An example of this philosophy is given in Table 4, which shows the possible instructions used to push an integer onto the parameter stack.

Table 4

range	instruction	bytes
x<-65537	*movl*	7
-65536<x<-257	*cvtwl*	5
-256<x<-64	*cvtbl*	4
-63<x<-1	*mnegl* (literal)	3
x=0	*clrl*	2
1<x<63	*movzbl* (literal)	3
64<x<255	*movzbl*	4
256<x<65535	*movzwl*	5
65536<x	*movl*	7

There also exists an alternate type of CONSTANT called a PARAMETER, which when compiled, has the same effect of compiling the minimal instruction that will push the integer on the parameter stack. PARAMETERs also use the IN-LINE bit of the header field. It was decided not to implement CONSTANT this way, as then n, the value of a CONSTANT, could not be changed by storing to the address returned by ' n.

The Future

Several new facilities are currently being developed. They are:

1. A Full Forth interface to UNIX signals.
2. A SAVE word which will save an image of the currently running Forth process in an executable file.
3. A method of dynamically loading and interfacing to subroutines written in other languages. The hardest aspect of this in a UNIX environment is the interaction between *malloc( )*, the standard memory allocation subroutine, and the Forth word FORGET.
4. Support of Forth-83.

References

1. Hammond, 1975. *Forth Programming System for the PDP-11.* DECUS No. 11-232, Decusers Society.
2. Stevens, W.R. 1979. *A FORTH Primer.* Kitt Peak National Observatory, Tucson, Arizona.
3. Sebok, W.L. 1981. *Use of an Automated Photographic Object Detection System to Analyze the Effect of Magnitude on the Angular Correlation Function of Galaxies.* California Institute of Technology, doctoral thesis.
4. Sebok, W.L. 1980. *SPIE Applications of Digital Image Processing to Astronomy,* 264, 213.
5. Sebok, W.L. 1984. "The Angular Correlation Function of Galaxies as a Function of Magnitude," *Astrophysical Journal Supplements.* Submitted.
6. McCourt, M. and Marisa, R.A. 1981. Forth Dimensions, III/4, 121.

Acknowledgements

The author would like to credit discussions with Wayne Hammond of the Jet Propulsion Laboratory for many of the ideas incorporated in this Forth implementation. I was an early user of his Forth compiler for a VAX under VMS. The details and the coding of the implementation discussed here are my own.

Manuscript received June 1984.