Forth language

Forth consists of words and new words must be compiled and entered into its dictionary. Following a description of the dictionary and compiler, Brian Woodroffe discusses advanced concepts in this third article.

by B. Woodroffe

Having shown how the address interpreter executes lists of addresses to execute program commands and that threaded code is compact, I will now explain how Forth builds these lists, i.e., how it compiles. Each list representing an action is rather like a verb in a natural language and in Forth is called a WORD. The collection of these words, which is Forth, is known as the dictionary. The outermost WORD in Forth breaks input text down into character strings which it then searches for in the dictionary. (Spaces are important, for instance, '−1' is treated as a negative number, whereas '− 1' is treated as the arithmetic subtraction operator followed by the positive number one.) If the string is found, it is executed, otherwise an error message is generated. The dictionary also needs a mechanism to allow the search to occur. Searching involves a traverse of all the words in the dictionary. Each entry has a pointer to the previous one (link field), which makes the dictionary a linked list. To enable matching of the source text each word also has its name in ASCII form (name field). Dictionary entries for each word, List 1, have four fields – name field, link field, code field and parameter field. The name field also contains data for use by the compiler (precedence and smudge bits) as will be explained, and it includes the length of the name to allow variable name lengths of up to 31 characters.

For language expansion it is important to be able to build dictionary entries for new words. This is done by invoking the compiler. When the compiler is invoked (Forth word ':'), the language state is switched from execution to compilation. Next, input text is scanned forward for the next text string which is used to build a newly created name field. The name is 'smudged' so that during the building of the incomplete definition, the same name cannot be found. This normally prevents recursion, but again in Forth, this rule can be overcome, List 2. Then the linked list of the dictionary is updated by copying it into the dictionary link and the address of DOCOL is copied as this new word's code field. Next, input text is scanned for character strings. As these character strings are matched with words that already exist in the dictionary, the code field of each word found is copied into the parameter field of the word being compiled. Finally, as the word to terminate compilation is encountered ';' SEMIS is copied as the last word of the definition and the Forth program is returned from the compile state to the execution state. The compiled word is now 'unsmudged' to allow it to be accessed.

The compile process can be quite long as many dictionary searches have to be made. As the dictionary is a linear list and the code routines which ultimately have to be compiled are at the bottom, it is a long search. No speed up algorithms such as hashing have been applied to standard FIG Forth though there has been experimentation[8,9,10]. As so much work is done during the compile phase the execution performance of newly defined words is nearly as quick as predefined words. Further, as the first half of the dictionary entry

List 1. Each dictionary entry has four fields called name field, link field, code field and parameter field.

```
        Dictionary entry
        7 6 5 4 3 2 1 0
NFA     1 p s < 1 e n >   p=precedence, s=smudge, len=length of name
        0 a s c i i 1
        0 a s c i i 2     ASCII characters of word
        0 . . . . . .
        1 a s c i i n     d7 set on last character of name
LFA     <             >   16 bit address of previous n.f.a.
CFA     <             >   16 bit address of code routine
PFA     <             >
        <             >   parameters, normally other c.f.as
        <             >
```

List 2. Example of recursion in Forth to calculate a factorial.

```
First define
: MYSELF
    LATEST              ( put address of word currently being defined on stack )
    PFA CFA ,           ( convert to code-field address and compile )
                        ( it in dictionary so that it may call itself )
  ; IMMEDIATE           ( as this word is to execute when in the compile state it has
                        'precedence'. )
Then use myself in the recursive definition
    : FACTORIAL ( n... )
      DUP 1 = IF ELSE   ( end of recursion?, yes leave 1 as 1!=1 )
      DUP 1 −           ( n, n−1... )
      MYSELF            ( call myself to calculate n−1! )
      *                 ( n!;=n*[n−1!] )
      ENDIF
    ;
```

List 3. Definitions of IF, ENDIF and ELSE.

```
: IF
  COMPILE 0BRANCH      ( compile into the dictionary the c.f.a. of '0BRANCH' )
  HERE                 ( place on stack where we are )
  0,                   ( make space for jump offset )
; IMMEDIATE            ( make compiler execute this word, even if in compile state )
: ENDIF
  HERE                 ( where are we? )
  OVER −               ( calculate offset to HERE executed in IF )
  SWAP !               ( patch in offset to address left by IF )
; IMMEDIATE
: ELSE
  COMPILE BRANCH       ( compile run time address of routine to skip false statements )
  HERE 0 ,             ( make space for jump offset )
  SWAP                 ( get address of where IF was )
  [COMPILE] ENDIF      ( use ENDIF to fix address, ENDIF is immediate and to overwrite
                       that such that it is compiled )
; IMMEDIATE
```

List 4. Examples of VARIABLE and CONSTANT.

```
: VARIABLE
    <BUILDS        ( variable is a new parent word )
    ,              ( store in the p.f.a. the value that was top of stack )
    DOES>          ( start defining what offspring will do )
    ;              ( nothing – p.f.a. is storage location for an offspring of type VARIABLE )
: CONSTANT
    <BUILDS ,
    DOES> @        ( constants provide a constant value which has been )
    ;              ( stored in their p.f.a. )

0 VARIABLE ABC     ( ABC is an offspring with initial value 0 )
1000 CONSTANT K    ( K is an offspring of value 1000 )
```

(name and link fields) is only required during compilation for fixed applications where compilation is not required, these fields can be deleted. This dramatically reduces the memory requirements of the Forth system[11], and can be especially useful when the code will be placed in rom.

To enable the compiler action of Forth to do more than just that described above certain words need to execute even when the language is in the compile state. This gives the compiler the full capabilities of Forth. These words are generally involved in building control structures for the compiler (IF-ELSE-ENDIF, see List 3). These words have a precedence bit set which the compiler recognises when it matches the input text so instead of compiling its code field it executes it. In the case of IF the compiler compiles into the dictionary the c.f.a. (code-field address) of **0BRANCH** and advances the dictionary pointer to allow the as yet unknown offset to be placed. It also pushes this address onto the stack so that when the compiler encounters an ELSE or ENDIF statement it can calculate the offset back to the IF statement and store the offset there. This shows the power of Forth in that the computational ability of the language is available to the compiler and to the user. Further there are times when words that would normally execute (i.e. have precedence) need to be compiled (i.e. execution action delayed until the word currently being defined executes). This is done using the word [COMPILE]. Again, it is sometimes required to delay compilation of a word until the word that contains it executes. This is done using the word COMPILE.

Advanced concepts

With an idea of how the inner interpreter (address interpreter) and the compiler (text interpreter) work we can now move on to advanced concepts including extension, vocabularies and virtual memory. Forth is either in the compile state, when it finds words and copies their code-field addresses into the dictionary to form new entries, or in the execute state, when it executes each code-field address encountered. I have shown how certain immediate words can override the state, and can even execute in the compile state. It is also possible to overrule words which are declared as immediate and compile them, as in the case of ELSE which was described earlier.

In Forth, even the compiler can be modified. Not only can new compiler control structures be introduced but also new compiler words may be defined. Normally the programmer would have to rewrite the compiler but with this feature, known as extensibility, modification is relatively simple. It involves use of the words <BUILDS and DOES> to define a new class of words. The defining word defines words of this new class. Behaviour of the new defining word is determined by the words between <BUILDS and DOES>, i.e. when a word of the new class is defined, behaviour of the new word during compilation is determined by what comes between <BUILDS and DOES>. When a word of this new class executes, it executes the words following DOES>. To allow the parent class-defining word to access its offspring'(class-defining word to access its offspring (class-defined word), the parameter-field address (p.f.a.) of the latter is placed on the data stack. Two simple examples from the Forth compiler are VARIABLE and CONSTANT, List 4. These can easily be expanced to form arrays and tables. The word ':' is also a defining type. When offspring of ':' are executed they call the word DOCOL which decides how to execute their parameter field. An alternative to DOES> is used to define ':'; the assembler is invoked so that the parent-word execution field is machine code and not Forth but in other respects it is the same.

The major part of Forth is the dictionary, and to enable different problems to be solved in different areas of the dictionary each problem is given its own vocabulary. The dictionary may have many vocabularies alongside the normal basic set of FORTH, ASSEMBLER and EDITOR. Using vocabularies means that the same word may have different meanings, depending on which vocabulary is active. FIG-Forth has two active vocabularies – CURRENT and CONTEXT. The former is the one in which words are defined and the latter is the one which is searched first. All vocabularies are linked to FORTH (Forth's definition in Forth). Much debate is taking place on the subject of vocabularies concerning the subject of searching vocabularies[11 – 13].

Virtual memory

Memory is the most precious resource of a computer and although Forth makes very efficient use of it, there are still times when programmers wish it was infinite. Disc memory is much cheaper than semiconductor memory but it is also slower. By the concept of virtual memory, the memory space available to the programmer is expanded beyond the main memory to include disc storage so memory capacity as far as the programmer is concerned is only limited by the capacity of the disc. In Forth, the virtual-memory cancept is only applied for data whereas in most processor applications (e.g INS16000 series) it is also applied for program storage. Through use of the word BLOCK, the programmer can

Forth words used in last month's program example

Stack operators

DUP pushes the top of the stack on to itself so the stack increases by one and the top and second stack elements are equal.

SWAP interchanges the two elements at the top of the stack.

OVER places a copy of the second element in the stack on the top. The original top element is now second and all other elements move down one.

ROT takes the third element and makes it the top element so the old top element becomes second and the old second element becomes third.

Arithmetic operators

MINUS replaces the top stack element with its two's complement.

+ takes off the top two elements, adds them and pushes back the result which becomes the new top of stack.

– takes off the top two elements and pushes back the result of second element minus the top element.

* takes off the top two elements, multiplies them and pushes back the result.

/ takes the top two elements, divides the second by the first, and pushes back the result.

n pushes the value of n on the stack.

SQRT takes the top two elements, treats them as double precision (32-bit), and pushes back the 16-bit square root.

0< if the top stack element is less than zero it is replaced by a one (true), if it is not less than zero it is replaced by a zero (false).

Control operators

IF tests and deletes the top stack element and executes the next word if the condition is true. If the condition is false, control skips to ELSE or ENDIF, whichever comes first.

ELSE marks the end of the IF-TRUE clause and the beginning of the IF-FALSE clause.

ENDIF is the end of an IF clause.

Terminal operators

CR sends a carriage return and line feed signal to terminal.

." starts text entry into the dictionary (until"). When executed it will type out the text string.

. converts the top of the stack to an ASCII string and types it out.

visualise the disc memory as processor memory. The Forth operating system recovers data from disc and places it in a buffer to make it accessible to the user program. Many such buffers exist in the host processor and BLOCK uses an algorithm that determines whoch blocks should be maintained in the store and which should be written back to the disc. With the right algorithm the number of disc accesses will be minimal and the apparent memory-access time low.

List 5. Representation of algorithm used for benchmark test, see Table 1.

```
8190 CONSTANT SIZE
0 VARIABLE FLAGS SIZE ALLOT            ( allocate 8191 bytes for an array )
: DO-PRIME
  FLAGS SIZE 1 FILL                    ( fill array with '1', <true> )
  0                                    ( counter )
  SIZE 0 DO                            ( set up a DO loop of 8190 times )
  FLAGS I + C@                         ( I is loop counter, get relevant flag )
  IF                                   ( C@, C! are byte versions of @,! )
             I DUP + 3 + DUP I +       ( stack is . . . count, prime, K )
    BEGIN                              ( begin a block )
      DUP SIZE <                       ( array index < size ? )
    WHILE                              ( test flag, to see if exit block )
    0 OVER FLAGS + C!                  ( set relevant flag false, FLAGS[K] )
    OVER +                             ( K:=k+prime )
    REPEAT                             ( end of block, loop back )
  DROP DROP 1+                         ( delete prime, K; one extra prime found )
  THEN                                 ( end of IF )
LOOP                                   ( end of DO . . . LOOP block )
. ." primes" ;                         ( print number of primes )
```

Space and time

I have shown how the Forth dictionary is created (i.e. its form), how it may be extended by compiling and how any processor may readily simulate the virtual Forth machine by means of indirect threaded code. As mentioned earlier, by introducing the concept of threaded code, execution speed is traded for code space. So one can expect that Forth is not as fast as the host processor's own code although it may approach it where many subroutine calls are made. Execution of a process defined in the source language divides into two parts; one is the examination of source text to find out what action is to be taken and the second is execution of the actions by the processor. In most systems the first part is carried out by compiling source text in the machine code of the host machine. In Forth this means compiled into the thread, the machine code of the hypothetical machine. Target machine code is then run in Forth using the address interpreter. Running time can be traded for space by choosing an intermediate language of suitable complexity. Running time performance of compilation has no effect on the running time of the application program, which leads to the view that the compiler should do as much work as possible. Unfortunately, compiling to machine code using a simple processor with limited addressing capability, such as a current 8-bit microprocessor, often results in the code not fitting into the memory so an intermediate target code is chosen, with the accompanying penalty of interpreting it. Forth's address interpreter costs some tens of percent.

Other losses occur because microprocessors are not zero-address devices so the zero-address function has to be simulated. Memory-space benefits are illustrated by the amount of memory required for a Forth system, which is typically 8Kbyte (may be rom) for virtual-machine simulation, the Forth compiler, i/o drivers, etc., and 8K for stacks, virtual-memory buffers and the user dictionary.

Forth is also fast because of the explicit use of the stack. In languages using the assignment operator, data normally resides outside the stack. It is brought to the stack, operated on, and finally placed back into the store. If the next statement uses the same variable it is once again taken from the store and placed on the stack. When computing partial results this causes excess memory traffic. Unless optimization is used this redundant memory activity will cause delays. Forth avoids this because normally data only resides on the stack. No unnecessary memory space or time is taken up by temporary variables.

It is interesting to compare Forth's performance with the commonly used language for microprocessors, Basic. Systems using Basic have little compiler action, the source text being saved in memory, although the key words are converted into internal tokens. During program execution, each token is parsed and acted upon in turn so the source of Basic's execution-time interpreter is close to that of the source text whereas Forth's source for the running-time interpreter is close to the language of the host computer. As all the work in a Basic system is done while the program is running the speed penalty is high, usually at hundreds of percent. Further, since Forth compresses object code into 16-bit addresses (code-field addresses are the equivalent of tokens) it is as efficient as Basic in terms of memory space.

Processing speed is an emotive issue without benchmark tests and unbiased benchmarks are notoriously difficult to produce. Table 1 was derived using the Seive of Eratosthenes (see List 5) and seems fair[14]. Qualitatively, it confirms what one could expect – assembly-language is faster, followed by compiled languages with interpreted Basic well behind. The table also shows how well the 6809 compares with newer and more popular designs and that it compares with at least one 16-bit device, the 8086. I would attribute this to the instruction set as was shown in the analysis of Forth word NEXT. A more elaborate, special-purpose instruction set does not necessarily lead to a more effective processor. This has been shown in recent research into reduced instruction set computers.

continued on page 61

Table 1. Relative speeds of various processor and languages.

Processor/language	Time in seconds
CRAY-1 Fortran	.11
68000 assembly language (8MHz)	1.12
PDP11/70 C	1.52
VAX 11/780 (C/Fortran/Pascal)	1.5-5.0
8088 assembly language (15MHz)	4.0
6809 assembly language	5.1
PDP11/40 C language	6.1
Z80 assembly language (4MHz)	6.8
6809 IMS Pascal compiled (2MHz)	8.9
PDP11/70 Decus Forth	11.8
Z80 Microsoft Basic compiler	18.6
8088 Pascal (Softech compiler) (15MHz)	19.4
68000 Forth (8MHz)	27
6809 FIG Forth (2MHz)	45
8088 FIG Forth (15MHz)	55
8086 FIG Forth (12.5MHz)	64
6809 FIG Forth (1.5MHz)*	67
Z80 Forth (Timin) (4MHz)	75
Z80 Forth (Laboratory Microsystems) (4MHz)	78
Z80 FIG Forth (4MHz)	85
6809 IMS Pascal P-code (2MHz)	105
6809 Basic 09 (2MHz)	238
6502 FIG Forth (1MHz)	287
6809 TSC Basic (2MHz)	830
Z80 Microsoft Basic	1920
APPLE integer Basic	2320
TRS80 Microsoft Basic	2250
6809 Computerware Basic	4303

* Used in my design as described in *Wireless World*

continued from page 55

List 6. Array boundary checking using <BUILD . . . DOES>.

```
: ARRAY                                   ( low high . . assumes low <high )
 <BUILDS
 OVER – SWAP OVER                         ( delta low delta )
 SWAP , ,                                 ( store low is p.f.a., delta as p.f.a. +1 )
 DUP + ALLOT                              ( that much storage, byte address machine )
 DOES>
 DUP ROT                                  ( . . p.f.a. p.f.a. index )
 SWAP @ – DUP 0<                          ( . . p.f.a. required-delta flag )
 IF ." array bound error, too low" QUIT THEN
 OVER 2+ @                                ( . . p.f.a. reqd allowed )
 OVER <                                   ( . . p.f.a. reqd flag )
 IF ." array bound error, too high" QUIT THEN
 DUP +                                    ( word index to byte index )
 + 4 +                                    ( add index-skip parameters, leaving array
                                            address )
;
```

Forth problems

So far, only advantages of Forth have been discussed but it has some disadvantages. The most obvious of these is notation. For the beginner, reverse-Polish notation and the lack of an assignment operator (:=) are considerable problems. Practice lessens the problems though program comments and stack diagrams generally remain necessary to show what is going on.

Floating-point arithmetic is not standard and all data manipulation assumes 16-bit two's-complement arithmetic, but it may be programmed in[15]. This shows Forth's origin in the control field of computing. However, many Forth programmers maintain that most problems can be reduced to scaled-integer arithmetic. This drawback makes one aware of the processing cost of floating-point arithmetic. Forth does not use 'data typing'. This means that integer operations are used when logical operations are being performed ('0=' for NOT). There are also separate operators for 32-bit arithmetic. Computer languages can usually apply different operations for the same operator by data typing.

A more serious drawback is the lack of built-in data structures – not that Forth is any worse in this respect than Basic or Fortran. What is lacking are the type of data structures available in Pascal. In common with the formerly mentioned languages, Forth lacks a method of checking for overflow and array boundary conditions in normal operation. But as shown in List 6. This can be programmed in. Naturally, this process increased execution time but when the application works a simpler version of array can be coded by missing out the check. Finally there is as yet no file management software. Access to disc information has to be done using BLOCK numbers.

Summary

I have shown that the programmer is released from the instruction set of the host computer with little time penalty by applying threaded code. Using the compiler, one may easily extend the Forth instruction set to suit one's own application. As the whole dictionary is available all of the time (ranging from virtual-machine instructions to <BUILDS . . . DOES> structures) the programmer can tackle low or high-level problems, such as i/o driving or word processing, with equal ease and efficiency. The consistent nature of the compiler and text interpreter allow easy interactive testing of code before it is compiled. Reverse-Polish notation simplifies the compilation process and allows it to be completed in one pass in a small memory. Virtual memory and vocabularies further enhance Forth by offering infinite data space and better control of the application software respectively. However, shortcomings of the language may prevent it from being applied to larger computers where its space-saving features are less useful. But it will continue to find many applications in small and interactive systems and real-time applications including hardware simulation, video games and test-equipment control.

References

8. M. McNeil, A hashed dictionary search method, FORML '81 papers, FIG.
9. T. Dowling, Hash encoded Forth Fields, FORM '81 papers, FIG.
10. K. Schleisiek, Separated heads, *Forth Dimensions*, vol. 12, no 5, pp. 147-150, FIG.
11. D. Petty, Utilizing vocabularies, FORML '81 papers, FIG.
12. J. Cassady, Towards a 79-Standard Fig-Forth Romable Vocabulary and Smart FORGET, FORML '81 papers, FIG.
13. G. Shaw, Executable Vocabulary Stack, FORML '81 papers, FIG.
14. J. Gilbreath, A. high-level Benchmark, *Byte*, Sep. 1981, pp.180-196 (for algorithm and performance data see *Byte*, Jan. 1983).
15. M. Jesch, High-level floating point, Forth Dimensions, vol.IV, no. 1, pp.6-12.