Should VARIABLE be an Immediate State-sensitive Word?

Steven M. Lewis

Department of Bioengineering
University of Southern California
Los Angeles, CA 90089

Abstract

We present an alternative to the word `VARIABLE`. In the absence of special modifiers, the word returns the contents. Previous use of modifiers allow the word to store or otherwise modify its contents. Unlike previous implementations the word is immediate and modifiers act at compile time allowing run time performance equivalent or superior to that seen with `@` and `!`. The extension of this concept to arrays is also presented.

Introduction

Forth, despite the clarity with which it handles many concepts, suffers from the clumsiness with which `VARIABLE`s are handled. The expression which in most languages would be written `C = A + B`, in FORTH is `A @ B @ + C !`. The latter is cryptic, requires additional keystrokes and is generally clumsier. To correct this deficiency a number of solutions have been proposed.

Bartholdi (1,2) proposed the `TO` concept, reacting to a suggestion by Charles Moore. He recognized the truism that most `VARIABLE`s are fetched much more often than they are used for other operations. He thus argued that a fetch should be the default action of a `VARIABLE` and other operations should require special treatment. His proposal was that a flag be maintained. The word `TO` would set the flag true. `VARIABLE`s would test and clear the flag. If the flag was false, the operation would be a fetch. If true, a store. Thus the example above would be written `A B + TO C` a considerable improvement in clarity.

The `TO` concept, as originally stated, suffers in that a severe penalty is imposed at run time, since `VARIABLE`s must test the state of the flag at each call. Rosen (3,5) proposed instead of `TO`, that FORTH words should have multiple code field addresses. This concept has been expanded by Schleisiek (7). `VARIABLE`s would thus compile a different CFA depending on the state of the `TOFLAG`, compiling words equivalent to `DOCON` (the operative portion of `CONSTANT` in many systems), `DOVAR` or an equivalent storage word. This solved the problem of speed at the expense of some fairly radical changes in the interpreter and, because the multiple CFA's must be in code, a large amount of system specificity. In addition, while adding 2 or 3 CFA's to `VARIABLE` will not require much additional memory, the storage increases as the number of operations supported increases.

This paper presents a solution which offers speed, flexibility and remains within the 83-Standard (6). The solution is general enough to handle arrays, double precision and floating point variables. It also allows the use of many of the automatic storage concepts used in the C language. `VALUE` is used in place of the word `VARIABLE`. It is made an immediate, state sensitive word which will either execute or compile the code for an indicated operation. The word `POINTER`, which allows the location operated on to be set at run time, has also been implemented.

Basic Implementation

A multivalued flag TOFLAG is defined in screen 0. A number of IMMEDIATE words are defined which set TOFLAG to various values. Table 1 shows the actions that have been included in this implementation. The C language contains a number of additional possibilities. It is necessary to leave the default value of TOFLAG a fetch. For consistency, it is recommended that 1 should cause a store and 2 cause the address of the indicated location to be left on the stack. Beyond that, it is only necessary that an implementation be internally consistent. The word : and all VALUE words (see below) will set TOFLAG to 0. Words which alter TOFLAG should be implemented directly before the affected VALUE. For example := X will cause the top of the stack to be stored in the VALUE X.

The VALUE creates an immediate, state-sensitive word. When executing, a VALUE word will perform the indicated action. When compiling, VALUE words will compile instructions to perform the action. Thus, if the TOFLAG is 0, a VALUE will compile the sequence (LIT) ADDR @. (Version 1, the faster version 2 is discussed below). If the TOFLAG is 1, the sequence (LIT) ADDR ! will be compiled. All testing of the flag is done at compile time leading to substantial improvements in run time performance.

The VALUE concept is readily adapted to arrays and other types of variables. The defining words for version 1 of VALUE assume only that the address of the object is on the stack. SCREEN 6 shows how easily this adapts to arrays. The use of VALUE() is as follows:

20 VALUE() MY_ARRAY creates the object MY_ARRAY
1 MY_ARRAY fetches the contents of element 1
21 3 := MY_ARRAY stores 21 in element 3 of MY_ARRAY

Table 1. Words which alter TOFLAG

Instruction	(TOFLAG **Value)**	**Action**
<default>	0	@
:=	1	!
%=	2	put affected address on stack
+=	3	+!
++	4	[a] --> [a]+1
--	5	[a] --> [a]-1
*=	6	[a] --> data*[A]
-=	7	NEGATE data then +!
&=	8	put address of object on stack (see text)
->P	100	store stack into pointer (see text)

Implementation Details

When any VALUE class word executes, the DOES> portion leaves an address on the stack. In execution mode, this address is used immediately. In compile mode, the literal instruction and the address are compiled to place the address on the stack at run time. (TO_ACTION) is a list of operations to perform on the data. TO_ACTION will fetch the indicated action and either execute the word or compile it for action at run time. One exception is made for TOFLAG = 2, to leave the address of the element on the stack. Instead of compiling a NOOP, nothing is compiled since the address is already on the stack.

The VALUE() code is similar to VALUE with two changes. First, the address placed on the stack is the base address of the array. The word VAL_ADDR takes an index and computes the address of the desired element. VALUE() executes or compiles VAL_ADDR. One exceptional case is treated. When TOFLAG is 8 (&=) then the base address of the array is the desired return and VAL_ADDR is not compiled. The code for the pointer takes the returned address and inserts a @ since the pointer holds the affected address. The case of TOFLAG being 100 (->P) indicating a store to pointer is treated as an exception and handled with the word PTR!.

It is important to put the := or another word which alters the TOFLAG immediately preceding the ARRAY since intervening VALUE class words will intercept the flag. The instruction &= will return an appropriate address for the entire array and is used when a word will operate on the array as an object. The instruction %= will leave the address of the indicated element on the stack.

The set of instructions in Table 1 can, with suitable modifications, be applied to objects addressing other than 2 byte words. The Word FVALUE has been included to demonstrate how := and %= could operate with floating point objects. Analogous operations to the remaining actions could be defined easily. They have not been included because the code would be dependent on details of the floating point implementation. The word assumes that F@ and F! work as expected and is independent of whether there is a separate floating stack or not.

One problem with VALUE and most other implementations of the TO concept is that, while it is possible to write a FORTH word which operates on a specific VALUE or array, it is not possible to write a word which operates on unspecified VALUEs without passing the address on the stack and giving up the convenience of the VALUE notation. To fulfill this function, data types POINTER and POINTER() are defined. A POINTER operates like VALUE except that instead of containing the affected data, the POINTER contains the address of the data. A special operation ->P causes an address to be stored directly in the data. Thus the code:

```
5 VALUE V1
POINTER P1
%= V1 ->P P1 ( store addr of V1 in P1 )
++ P1 ( effectively increments V1 )
```

A more useful concept is POINTER(), a pointer to an array. Once a POINTER() word is set to point at an array, it behaves in a similar manner to the array itself.

```
10 VALUE() MY_ARRAY ( create a 10 element array )
POINTER() P1        ( create an array pointer )
: INC_ARRAY ->P P1 -1 P1 0 DO I ++ P1 LOOP ;
                    ( increments all elements of the array on the stack)
                    ( note element -1 of a VALUE() is the size )
&= MY_ARRAY INC_ARRAY ( increment ALL elements of MY_ARRAY )
```

In dealing with arrays, double precision variables and floating point data, there is usually enough code involved with operating on the object to make the fetch and store portions a minor portion of the entire cycle. For single VALUEs, there is a significant penalty in accessing a VALUE as compared to a CONSTANT, because the CONSTANT will often use a CODE word to perform the fetch rather than placing its address on the stack and then fetching. Such code words will be quite system dependent. A general form may be obtained by looking at (LIT) or DOCON if the sources are available. In version 2 of VALUE, fetch and store operations are implemented as code words. Screen 11 lists the timing on a 68000 system running the Laxen-Perry implementation of the 83-Standard (No-Visible-Support software) under CPM-68k. Code words make the operation of VALUE run at the same speed as accessing a CONSTANT and considerably faster than accessing a VARIABLE. Storing into a VALUE runs at speeds comparable with fetching and appreciably faster than the VARIABLE store operation.

Discussion

VALUE offers several advantages over alternative implementations of the TO concept. Because all tests are done at compile time, there is no penalty at run time. If the CODE version is used, there may actually be time savings at run time. Additionally, the word is within the 83-Standard and requires no modification of existing FORTH words. It generates shorter code, generally easier to read. The use of a multivalued action flag allows VALUE words to be used in modes like increment, decrement and +!.

Pointers constitute a particularly powerful extension on the TO concept. While there is a small penalty at run time involved in the use of pointers, the convenience of writing general routines to operate on arrays and still being able to use TO class operations will usually justify any run time penalty.

Early papers (1) suggested that implementation of the TO concept would eliminate the need for @ and !. This seems unlikely. More likely @ ! and ' could be used more in lower level, systems programming. Inexperienced users could enjoy the advantages of FORTH without the necessity of dealing with these words.

&= is particularly useful. It is frequently necessary to pass an array to a subroutine. Using ' seems clumsy. Depending on the structure of the array, the address of the 0th element may or may not be appropriate. &= offers a way for all arrays to return a uniform address. &= is also needed to allow setting of pointers to arrays. If pointers are used extensively, for example, in routines where vectors are being manipulated, it may be more useful to have an array return the base address as the default and handle all accesses through pointers.

There are several difficulties and points to keep in mind. The current implementation assumes a 16 bit stack and would require minor modification to work on 32 bit systems. If the interpreter aborts between the time the TOFLAG is set and the time it is read and reset, TOFLAG will have an erroneous value at the first reference. While : has been modified to prevent this from affecting compiled code, there still may be errors at run time. If possible ERROR or some similar routine should be modified to clear the TOFLAG as well. VALUE adds another state sensitive word to the language. Users may find that the benefits gained by implementing the TO concept in this way offset possible confusion caused by this addition.

VALUE is a powerful, portable concept that I find increasingly useful in my FORTH programing. The flexibility of the approach encourages the programmer to add new functions to an extremely powerful concept.

I thank Chris Horne for his helpful comments and suggestions. This work supported in part by HL30711.

References

1) Bartholdi, Paul, "The 'TO' solution", *Forth Dimensions* Vol. 1, No. 4:38-40, 1979.
2) Bartholdi, Paul, " 'TO' solution continued..." *Forth Dimensions* Vol. 1, No. 5:48, 1979.
3) Rosen, Evan, "QUAN and VECT - High Speed, Low Memory Consumption Structures", *FORML Conference Proceedings*, 191-197, 1982.
4) Dowling, Tom, "The Quan Concept Expanded", *Proceedings 1983 Rochester Forth Conference*, 89-92, 1983.
5) Perkel, Marc, "The Integer Solution", *Forth Dimensions* Vol. 6, No. 2:18-19, 1984.
6) *Forth-83 Standard*, Forth Standards Team, 1984.
7) K. Schleisiek, "Multiple Code Field Data Types and Prefix Operators", *Journal of Forth Application and Research*, Vol. 1, No. 2:55-64, 1983.

Manuscript Received December 1984.

```
Scr # 0        VALUE1.BLK
\ VALUE AN IMPLEMENTATION OF THE TO CONCEPT

VARIABLE TOFLAG       \ will be flag for value operation

: CLRTO 0 TOFLAG ! ;  \ because we do this a lot

\ words to set TOFLAG see table 1 for actions
: := 1 TOFLAG ! ;   IMMEDIATE    : %= 2 TOFLAG ! ;  IMMEDIATE
: += 3 TOFLAG ! ;   IMMEDIATE    : ++ 4 TOFLAG ! ;  IMMEDIATE
: -- 5 TOFLAG ! ;   IMMEDIATE    : *= 6 TOFLAG ! ;  IMMEDIATE
: -= 7 TOFLAG ! ;   IMMEDIATE    : &= 8 TOFLAG ! ;  IMMEDIATE
: ->P 100TOFLAG ! ; IMMEDIATE  ( special code for POINTERS )

: : CLRTO [COMPILE] : ; IMMEDIATE ( make sure TOFLAG clear )


Scr # 1        VALUE1.BLK
\ TESTS FOR RANGE AND VALUE

8 CONSTANT MAX_TO     \ maximum defined operations

: TEST_TO  ( n--n make sure n ok for CASE )
   DUP 0< OVER MAX_TO > OR
   IF ABORT" ILLEGAL TO VALUE " THEN ;

\ for convenience in testing the most common cases
: TO0 TOFLAG @ 0= ;        ( <>--f T if flag = 0 )
: TO1 TOFLAG @ 1 = ;       ( <>--f T if flag = 1 )
: TO2 TOFLAG @ 2 = ;       ( <>--f T if flag = 2 )
: NOT_TO8   TOFLAG @   8 = NOT ;  ( <>--f T if flag not 8    )
: NOT_TO100 TOFLAG @ 100 = NOT ;  ( <>--f T if flag not 100 )


Scr # 2        VALUE1.BLK
\ A SERIES OF WORDS TO IMPLEMENT THE REQUIRED OPERATIONS

: (++) ( addr--<> ) 1 SWAP +! ;  \ note a good word for code

: (--) ( addr--<> ) -1 SWAP +! ; \ another good word for code

: (*=) ( data,addr--<>)          \ [addr]=[addr]*data
     SWAP OVER @ * SWAP ! ;

: (-=) ( data,addr--<>)          \ [addr]=[addr]-n same as -!
     SWAP NEGATE SWAP +! ;

( make a state sensitive word to handle setting pointers )
: PTR! ( executing N,Addr--<>, compiling <>--<> )
     STATE @ IF COMPILE ! ELSE ! THEN
     CLRTO ;
```

```
Scr # 3        VALUE1.BLK
\ TO_ACTION the operative portion of VALUE
\ NOTE a CASE statement could be used for much of this
\ TO_ACTION is the crude body of a CASE statement
CREATE (TO_ACTION) ] @ ! NOOP +! (++) (--) (*=) (-=) NOOP [

' (TO_ACTION) >BODY CONSTANT &ACTION \ get address to use

: TO_ACTION ( compiling Addr--<> executing flag dependent )
     TOFLAG @ TEST_TO       \ get flag and test range
     2* &ACTION + @         \ compute desired action
     STATE @ 0= IF          \ if executing do it, if compiling
          EXECUTE           \ compile it
     ELSE
          TOFLAG @ 2 = NOT          \ expect addr so action 2
          IF , ELSE DROP THEN       \ will do nothing
     THEN CLRTO ;                   \ RESET TOFLAG

Scr # 4        VALUE1.BLK
\ VALUE version 1 and POINTER

: VALUE  ( creat N--<> does> flag dependent )
   CREATE , IMMEDIATE ( N--<>  must be initialized )
   DOES>
          STATE @ IF
              COMPILE (LIT) ,  THEN  \ put addr on stack
          TO_ACTION ;                \ do operation

: POINTER  ( create <>--<> does> flag dependent )
   CREATE 0 , IMMEDIATE ( <>--<> note must be set before use)
   DOES>
          STATE @ IF
               COMPILE (LIT) , NOT_TO100 IF COMPILE @ THEN
          ELSE NOT_TO100 IF @ THEN THEN
          NOT_TO100 IF TO_ACTION ELSE PTR! THEN ;

Scr # 5        VALUE1.BLK
\ VALUE() A TO CONCEPT ARRAY

: VAL_ADDR ( INDEX,BASE--ADDR compute address of array elem.)
   SWAP 2* + 2 + ;

( NOTE TOFLAG = 8 SAYS use base addr not indexed addr )
( Note element -1 of VALUE() is the array size )
: VALUE()   ( create Size--<> does> flag dependent  )
   CREATE DUP , 2* ALLOT IMMEDIATE  ( S N--<> NOTE  IMMEDIATE)
      DOES>   ( INDEX--?? depends on TOFLAG )
          STATE @ IF
               COMPILE (LIT) ,   ( put base addr on stack)
               NOT_TO8 IF COMPILE VAL_ADDR THEN
          ELSE NOT_TO8 IF  VAL_ADDR THEN THEN
          TO_ACTION ;   \ in either state addr is on stack

```

```
Scr # 6         VALUE1.BLK
\ POINTER() A POINTER TO AN ARRAY

: POINTER()    ( create <>--<> does> flag dependent )
    CREATE 0 , IMMEDIATE  (S <>--<> note must set before use)
    DOES>    ( index--?? depends on toflag )
           STATE @ IF
                COMPILE (LIT) ,
                NOT_TO100 IF  COMPILE @
                    NOT_TO8   IF  COMPILE VAL_ADDR  THEN
                THEN
           ELSE
                NOT_TO100 IF @
                NOT_TO8   IF VAL_ADDR THEN
                THEN
           THEN
           NOT_TO100 IF TO_ACTION ELSE PTR! THEN ;
```

```
Scr # 7         VALUE1.BLK
\ FVALUE handles only @, ! and %= cases

: FVALUE   ( create Float--<> does> flag dependent )
   CREATE HERE F! 4 ALLOT IMMEDIATE      \ store initial value
   DOES>
   TOFLAG @ DUP 2 > IF ABORT" ILLEGAL TOFLAG" THEN
   STATE @ 0= IF
          TO0  IF F@ ELSE             \ handle @
          TO1  IF F! THEN THEN        \ default to address
      ELSE
          COMPILE (LIT) ,            \ cause addr to stack
          TO0  IF COMPILE F@  ELSE   \ COMPILE @
          TO1  IF COMPILE F!  THEN  THEN   \ ! DEFAULT TO ADDR
     THEN CLRTO ;                   \ always reset TOFLAG


```

```
Scr # 8         VALUE1.BLK
\ REWRITING @ AND ! PORTIONS IN CODE  68000 CODE

CODE (VALUE@) \ same as DOCONSTANT )
     IP )+ D7 MOVE      \ on 68000 this is needed to prevent
     D7 W LMOVE         \ the sign bit from extending
     W ) SP -) MOVE NEXT C;

CODE (VALUE!) ( does ! instead of @ )
     IP )+ D7 MOVE      \ 68000 and implementation specific
     D7 W LMOVE         \ just needed to fix extension
     SP )+ W ) MOVE NEXT C;





```

```
Scr # 9        VALUE1.BLK
\ FAST_TO_ACTION uses code words for @ and !

: FAST_TO_ACTION  \ will compile (VALUE@) and (VALUE!)
    TOFLAG @ TEST_TO    \ get flag and test range
    2* &ACTION + @      \ compute desired action
    STATE @ 0= IF
         EXECUTE        \ executing has ADDR,ACTION
    ELSE                \ compiling has ADDR,ACTION
         SWAP           \ we must compile addr first
         TO0 IF COMPILE (VALUE@) , DROP ELSE   \ HANDLE @
         TO1 IF COMPILE (VALUE!) , DROP ELSE   \ HANDLE !
         COMPILE (LIT) ,      \ other cases need the addr
                               \ on the stack
         TO2 IF DROP     \ expect ADDR so action 2 is nothing
         ELSE , THEN THEN THEN     \ otherwise compile action
    THEN CLRTO ;        \ reset TOFLAG
```

```
Scr # 10       VALUE1.BLK
\ VALUE VERSION 2. this version uses code words for @ and !

: VALUE
    CREATE , IMMEDIATE ( N--<>  must be initialized )
    DOES>
         FAST_TO_ACTION ;   \ FAST_TO_ACTION does operation

\ note FAST_TO_ACTION  is state sensitive
```

```
Scr # 11       VALUE1.BLK
\ TESTS AND EXAMPLES
0 VALUE    AVAL          \ test VALUE
VARIABLE   AVAR          \ test CONSTANT
0 CONSTANT ACON          \ test VARIABLE
\ in each case perform a million operations

( 36 sec version 1; 25 sec version 2; 23 sec Constant )
: @TEST_VAL 1000 0 DO 1000 0 DO AVAL DROP LOOP LOOP ;
: @TEST_CON 1000 0 DO 1000 0 DO ACON DROP LOOP LOOP ;

( 37 sec version 1; 25 sec version 2; 37 sec variable )
: !TEST_VAL 1000 0 DO 1000 0 DO 1 := AVAL LOOP LOOP ;
: !TEST_VAR 1000 0 DO 1000 0 DO 1 AVAR  ! LOOP LOOP ;

( 10 sec to subtract out irrelevant structure )
: EMPTY-LOOPS 1000 0 DO 1000 0 DO LOOP LOOP ;
```