EuroForth 2020
Extending the VALUE concept

Abstract

For many years, variables that return their addresses have been a standard part of Forth. Some years ago, an alternative concept of values, that return their contents, was introduced. This has proved to be so useful, that we have extended the concept to elements of different size, arrays and structures.

N.J. Nelson B.Sc. C. Eng. M.I.E.T.
Micross Automation Systems
Unit 6, Ashburton Industrial Estate
Ross-on-Wye, Herefordshire
HR9 7BW UK
Tel. +44 1989 768080
Email njn@micross.co.uk

1. Introduction

We were all brought up with VARIABLEs, and for many years we were perfectly happy with them. Their operators @ (fetch) and ! (store) were placed after the variable, in the proper Forth-like way. Then along came VALUEs, which turn out to be rather useful, especially if you think about them in a Forth-like way too. It turns out that they get really very useful if their concept is extended to other types of data storage.

2. Why VARIABLEs are not so great

It's not VARIABLEs themselves, it's their operators. If @ and ! were used *only* for VARIABLEs, that would be fine. But they're also used to access elements in structures and arrays. And the problem is, you don't feel you know exactly what they do. Actually, what you know is, they fetch and store data that is the same size as the version of Forth (note: not necessarily the size of the operating system). But what if you are trying to write code that works in different sizes of Forth? Suddenly there are horrible problems. The access of structure elements of fixed sizes doesn't work any more.

3. Why VALUEs are great

I list the reasons from least important to most important.

a) VALUEs return their content, not their address. It is easy to observe in most applications that there are far more fetches than stores. Therefore, using a VALUE instead of a VARIABLE results in a small simplification of the code. There is a direct correlation between code complexity and bug infestation! So, VALUEs should result in fewer bugs.

b) VALUEs are initialised at compilation time.
In theory this saves explicit initialisation in the code, but in practice we find that one does not often know an appropriate initialisation value at coding time.

c) If all VARIABLES are replaced by VALUEs, you don't need @ and ! any more.
For those cases where you still need to fetch and store from arrays or structures, you can use words that specify the size that you need explicitly, such as C@, W!, I@, L! etc.

d) VALUEs work in just the same way as locals. Note that to avoid confusion, I never refer to "local variables" precisely because locals *do not* act like VARIABLEs. We are great believers in locals, because they improve code readability.

4. Operators or Modifiers?

VALUEs on their own, just returning their content, are quite happy in Forth. They behave just like CONSTANTs. However, when you need to store, or do other things to a VALUE, it seems at first to look un-Forth-like.

```
123 -> MYVALUE
```

The "operator" appears before the VALUE. To a brain steeped in Forth, this looks all wrong. However, if you decompile the result, you will find no sign of the word "->". What this has actually done is to modify the compilation action for the following VALUE.

As soon as you stop referring to words like "->" as operators, and start calling them "modifiers" instead, then all your Forth instincts are satisfied again.

As a further improvement, rather than fill our code with "magic numbers" we have also used an enumeration to describe all the modifiers.

```
ENUM VALMODS {
  VMOD@        \ Fetch
  VMOD!        \ Store
  VMODADDR     \ Address
  VMODINC      \ Increment
  VMODDEC      \ Decrement
  VMOD+!       \ Add
  VMODOFF      \ Zero
  VMOD-!       \ Subtract
  VMODSIZE     \ Size of
  VMODSET      \ Set
};
```

5. Extending the concept to different data sizes

In my opinion, only one additional size is needed - for floating point values.
This is already provided, along with its local equivalent, in some compilers but not others.

```
1.2E3 FVALUE MYFVAL  ok
MYFVAL F. 1200.  ok
4.56E7 -> MYFVAL  ok
```

6. Extending the concept to arrays

In my opinion, this is where things get really useful.

a) VINDEX

I propose VINDEX to create an indexed array of values.

```
: VINDEX nsize "name" -- ;
Exec: (unmodified) nindex -- ncontents
```

All the modifiers that can be used with VALUES are available.

```
10 VINDEX MYINDEX  ok
12 1 -> MYINDEX  ok
34 2 -> MYINDEX  ok
1 MYINDEX . 12  ok
2 MYINDEX . 34  ok
1 SIZEOF MYINDEX . 4 ok
```

Note that the SIZEOF applies to the individual element, not to the number of elements in the array.

Because we have not found value initialisation to be very useful, no initial value is specified for VINDEX. All elements are initialised to zero.

There seems to be no consensus about whether array indices should be zero based (to keep programmers happy) or one based (to keep "normal" people happy). Therefore, my implementation of VINDEX actually creates space for both possibilities.

```
10 0 -> MYINDEX  ok
100 10 -> MYINDEX  ok
0 MYINDEX . 10  ok
10 MYINDEX . 100  ok
```

Because -> is a modifier, not an operator, then the statements
```
12 1 -> MYINDEX
```
and
```
12 -> 1 MYINDEX
```
have the same effect.
However, in practice, the index itself is often a VALUE, and the statements
```
12 MYVAL -> MYINDEX
```
and
```
12 -> MYVAL MYINDEX
```
do not have the same effect.
Therefore, we have started as good programming practice always placing the modifier immediately before the word that it is modifying, even when the index is not a VALUE.

An interesting possibility now presents itself. We have seen that incorrect calculation of an index is a very common software bug, and can be very hard to trace.

It is now easy to constrain the index value applied to a VINDEX, and report the error.

```
1 11 -> MYINDEX
Invalid index 11 for MYINDEX length 10
 ok
```

The error message is reported on the terminal window, if in debug mode, or added to a log file if in normal run mode.

b) VMATRIX

I propose VMATRIX to create a two dimensional indexed array of values.

```
: VMATRIX nsizex nsizey "name" -- ;
Exec: (unmodified) nindexx nindexy -- ncontents
```

This operates in exactly the same way as VINDEX but with two indices.

```
10 20 VMATRIX MYMATRIX  ok
12 3 4 -> MYMATRIX  ok
3 4 MYMATRIX . 12  ok
```

It also checks for valid indices:

```
1 10 21 -> MYMATRIX
Invalid index 10 21 for MYMATRIX length 10 20
 ok
```

7. String indices

Following on from the use of VINDEX, I realised that the same concept could easily be used to create arrays of strings.

```
: STRINDEX narraysize nmaxlen "name" -- ;
Exec: (unmodified) nindex -- addr
```

In this case, being a string, the use of the unmodified name simply returns the string address. However, for a nice compatibility, -> does the string store.

```
10 100 STRINDEX MYSTRS  ok
Z" abc" 1 -> MYSTRS  ok
Z" xyz" 2 -> MYSTRS  ok
1 MYSTRS Z$. abc ok
2 MYSTRS Z$. xyz ok
1 SIZEOF MYSTRS . 100  ok
```

A possible security enhancement, not yet implemented, would be to constrain the length of the string during a store, and report errors in the same way as an indexing error.

8. Structures with values

Forth structures can have elements of any size - byte, word, int, cell, string etc. Normally, the word that defines the element returns a calculated address. A frequent problem for the programmer is to use the correct operator (C@. W@ etc.) for the size of the element. Mistakes happen, are often disastrous and can be very hard to find.

It occurred to me that if the elements of a structure were like values, then this type of mistake could be eliminated.

```
: VFIELD structlen size "name" -- structlen' ; ??? -- ???
```

defines a value type field of arbitrary length.

The following words are added for all standard sizes:

```
: VBYTE        _BYTE     VFIELD  ; \ Value type byte field
: VWORD        _WORD     VFIELD  ; \ Value type word field
: VINT         _INT      VFIELD  ; \ Value type int field
: VLONGLONG    _LONGLONG VFIELD  ; \ Value type longlong field
```

A demonstration:

```
STRUCT MYSTRUCT
  VINT         my1
  VWORD        myword
  VBYTE        mybyte
  100  VFIELD  mystring
END-STRUCT

MYSTRUCT BUFFER: MYSTR


: TESTER
  123 MYSTR -> my1
  45  MYSTR -> myword
  67  MYSTR -> mybyte
  Z" qwerty" MYSTR -> mystring
  MYSTR my1 .
  MYSTR myword .
  MYSTR mybyte .
  MYSTR MYSTRing z$. SPACE
  MYSTR SIZEOF my1 .
  MYSTR SIZEOF myword .
  MYSTR SIZEOF mybyte .
  MYSTR SIZEOF MYSTRing .
;

tester 123 45 67 qwerty 4 2 1 100  ok
```

9. Dynamic creation of value arrays

In a previous paper, I explained how VALUEs were created automatically when an XML file, created by the GTK visual design program GLADE, was loaded. Creating VALUEs programmatically requires a different form:

```
: ZVALUE z$name  ival -- ; Exec: (unmodified) -- val
```

It is also very useful to be able to create arrays of different elements sizes programatically. Therefore, alternative forms for these are also provided, such as:

```
: ZVINDEX nsize z$name -- ; Exec: (unmodified) nindex -- ncontents
```

These are used when creating system configuration settings directly from an SQL table, as I will explain in another paper.

10. Future ideas

In my continued search for ways to make code more concise and readable, I am always looking for how the best features of other languages can be adapted to Forth.

The PHP language is particularly good in its use of the results of an SQL query. The column names of the query result are useable in the code.

It should be possible to do something similar in Forth by dynamically creating locals, and loading them automatically with the values of the results. All SQL results are in string form, and numbers have to be converted. But by looking up the data type of a column in the result, it should be possible to convert automatically.

It might look something like this:

```
MAXOPERATORS VINDEX   OPNUMS
MAXOPERATORS STRINDEX OPNAMES

: GETOPS ( --- ) \ Get operator names & nos. from SQL into memory
  SQL| SELECT opnum,opname FROM OPERATORS |SQL> IF    \ Run query
    PROWS 0 DOROW                                     \ Each row
      ropnum  I -> OPNUMS                             \ Save no.
      ropname I -> OPNAMES                            \ Save name
    LOOP
  THEN
;
```

Here I imagine that ropnum and ropname are automatically created locals, named by adding a prefix to the column name of the result. "ropnum" returns a number, because it has looked up the column type, and because it is a numeric type, it has done the conversion from a string. "ropname" returns an address because the column type is some sort of string.

As yet, I have not figured out how to create locals dynamically!

11. Conclusion

The concept of VALUEs instead of VARIABLEs is quite useful on its own. When extended to include arrays and structures, it becomes very useful indeed.