Technical Note

The QUAN Concept Expanded

Tom Dowling
Miller Microcomputer Services
61 Lake Shore Road
Natick, Massachusetts 01760

Abstract

The QUAN (multiple CFA words) was presented by Evan Rosen (1) at the fourth FORML Conference. This concept is very close to the "TO concept" but moves the selection of an EXECUTE routine from the execution to the compilation phase. The QUAN concept has been implemented in MMSFORTH V2.2 and extensions have been studied. The use of this type word for data hiding and implementation-independent applications will be investigated. Examples of the use of these concepts in practical situations will be presented. Using QUAN instead of VARIABLE leads to both time and memory savings. The resulting code is also easier to read with the removal of redundant @ and !. Experience with this concept in several projects proves the advisability of including it in the next FORTH standard.

The QUAN concept defines a class of words which have multiple CFA fields. The first CFA returns the value of the word, the second stores a value in the word, and the third returns the address of the storage location of the word. The first CFA is executed or compiled if the QUAN type word is not immediately preceded by IS or AT, the second CFA if it is immediately preceded by IS, and third CFA if it is immediately preceded by AT. IS and AT (BLOCK 18) are immediate words which FIND the next word in the input stream, check that it is a QUAN type word (the header has 2 bits to record the number of CFA's a word has), and compile or execute the respective CFA's. Since the decision of which CFA to use is done at compile time, there is no execution time penalty in using these words. As an example, if we compare the execute overhead for a QUAN X and a VARIABLE Y; for fetching the value X or Y @ ; the QUAN executes NEXT one time and the VARIABLE two -once for Y and once for @ ; for storing 5 IS X or 5 Y ! the same argument holds. Therefore, in these situations, the QUAN should be twice as fast as the VARIABLE. The same type argument holds for the storage used at each compilation of these words; the QUAN uses half the memory of the VARIABLE. Since the appropriate CFA is determined at compile time rather than at execute time (which is the case for the TO concept), the need for a TO stack for nested TO definitions is removed. We use IS instead of TO because of the confusion between 2 and TO which has occurred when reading code over the phone.

The QUAN concept is also useful for data hiding or zero address applications as documented in the references. We have used the QUAN concept for the support words for the 8087 floating point chip. This chip contains its own stack and performs floating point operations very quickly, but data movement into and out of the chip is slow. The floating and double-floating QUAN type words store and get their values from this stack. Complex type QUANs store and get two values from this

**This work originally appeared in the Proceedings of the 1983 Rochester Forth Applications Conference.*

stack. Long address type floating QUAN arrays remove the difficulty of addressing data beyond a 64K limit. Only the indexes of these arrays must be in the 0 to 64K limit with address translation performed by the words themselves. As an example: 255 255 2FQARRAY X defines a floating two-dimensional array with each index running from 0 to 255. This array requires 256K of memory: 256 * 256 * 4 bytes/element. The use of this word only requires the indexes to be on the stack: 200 125 X places the value of X(200, 125) on the 8087 stack, and 225 130 IS X fetches the value on the top of the 8087 stack into X(225, 130).

The QUAN concept is important for programs which use data which may be on the disk or in memory. Disk resident data requires the use of UPDATE when data is stored. This can be included in the IS CFA. Therefore, the algorithm code of the program can read the same for virtual or memory data, and only the code for the data definition is different.

A QUAN defining word has been developed and an implementation for the 8088 (BLOCK 19) is presented. This word uses any existing FORTH words, code or colon, to define the actions for value fetching, storing, and address calculation. The code segment following BUILD-QUAN is the storage allocation and initialization code.

```
QUAN-DEF QUAN @ ! NOP BUILD-QUAN 0 , ;
```

This defines the defining word QUAN. Words defined with QUAN will return their value when used alone, set their value when preceded by IS, and return their parameter address (BODY) when preceded by AT.

```
QUAN X QUAN Y QUAN Z 6 IS X 7 IS Y

X Y + IS Z  Z is now equal to 13.

: @EXECUTE @ 2- EXECUTE ;

QUAN-DEF VECT @EXECUTE ! NOP BUILD-QUAN ' NOP , ;
```

This defines the defining word VECT. Words defined with VECT will execute the word whose PFA is in their parameter field when used alone, set their value when preceded by IS, and return their parameter address (BODY) when preceded by AT.

```
VECT PLUS ' + IS PLUS  When PLUS is used it will execute +.
```

QUAN-DEF would not normally be used for these definitions, but they are presented to clarify the idea rather than as recommended implementations.

References

1. "QUAN and VECT - High Speed, Low Memory Consumption Structures", Evan Rosen, *Proceedings, Fourth FORML Conference* (FORTH Modification Laboratory), October 6-8, 1982, Asilomar, California.
2. "The 'TO' Solution", Paul Bartholdi, *FORTH Dimensions*, Vol. 1, no. 4, 1979, Forth Interest Group, P. O. Box 1105, San Carlos, California 94070.
3. " 'TO' Solution Continued ...", Paul Bartholdi, *FORTH Dimensions*, Vol. 1, no. 5, Forth Interest Group.
4. Letter by George Lyons, *FORTH Dimensions*, Vol. 1, no. 5, Forth Interest Group.
5. "The 'TO' Variable", Michael McNeil, *Proceedings, 1980 FORML Conference*, November 26-28, 1980, Asilomar, California.

BLOCK 18 [18 :0]

```
 0 ( 05/03/83 QMARK QCOMPILE IS AT )
 1
 2 : QMARK        LAST C@ 2 OR LAST C! ;
 3
 4 : QCOMPILE ?FIND SWAP ?TH ( ?-temp-head ) ?DUP 0= IF OVER THEN
 5              6 - C@ 3 AND 2* OVER <
 6              ?ABORT + STATE @ IF , ELSE EXECUTE THEN ;
 7
 8 : IS          2 QCOMPILE ; IMMEDIATE
 9 : AT          4 QCOMPILE ; IMMEDIATE
10
11
12
13
14
15
```

BLOCK 19 [19 :0]

```
 0 ( QUAN-DEF QUAN @ ! NOP BUILD-QUAN 0 , ; ... QUAN X 5 IS X )
 1
 2 : MAKE-CFA-PART ASSEMBLER
 3  # BX ADD BX PUSH # BX MOV [BX] JMP ;
 4
 5 CODE JMP-BUILD SI POP NEXT
 6
 7 : QUAN-DEF : 3 0 DO ?FIND 6 I 2* - MAKE-CFA-PART LOOP
 8  FIND IF QUESTION THEN
 9  DOES> CREATE QMARK -2 ALLOT 3 0 DO DUP , 9 + LOOP
10       JMP-BUILD ;
11
12 QUAN-DEF Q @ ! NOP BUILD-QUAN 0 , ;
13
14
15
```