

Editing Words In TurboForth

It is a feature of Forth that if a word is re-defined, any previously defined words that reference the changed word do *not* change to reflect the new definition.

This is often 'sold' as a powerful feature of Forth (and it does indeed have one very powerful use in particular) but in reality, it is simply a side effect of the the linked list nature of the dictionary, and ITC (indirect threaded code).

As colon definitions are entered, the dictionary grows towards higher memory addresses. One of the dictionary fields is a link to the previous word in the dictionary.

When the dictionary is searched for a word during compilation, the word `FIND` 'walks' along the dictionary, from the newest word to the oldest word. When it finds the word, it compiles the *address* of the word into the current colon definition.

It is the fact that the *address* of the word is compiled that constitutes the bulk of the problem. If a word is re-defined, it would require scanning the entire program from beginning to end, looking for instances of the *old* address, and replacing them with the new address. However, that is not as simple as it sounds. There is no way to determine (without some degree of de-compilation) if a cell with a matching value is actually a reference to the word in question, or some item of data that just happens to have the same numeric value.

Consequently, when a word is redefined in Forth, only *subsequently* defined words will take on the new definition (because they come *after* it in the dictionary).

For example:

```
: HELLO ." HELLO WORLD" CR ;  
: TEST HELLO ;  
TEST
```

The above displays HELLO WORLD

Now, change HELLO

```
: HELLO ." HELLO MOTHER" CR ;  
TEST
```

The above displays HELLO WORLD – not HELLO MOTHER.

In reality, when `HELLO` was “redefined” it wasn't “redefined” at all. A *new* dictionary entry, with the same name as the old one was created, but `TEST` still has the address of the *old* `HELLO` encoded within it.

To prove this, `TEST` can now be redefined:

```
: TEST HELLO ;  
TEST
```

And now we see HELLO MOTHER.

This is because TEST is simply a new entry in the dictionary. During compilation, it sees the 'new' definition of HELLO and compiles its address. The *old* definitions of both HELLO and TEST are still there, but essentially lost, since FIND will always find the most recent version of a word and use it.

The traditional way to get around this problem is to use FORGET to remove words from the dictionary, and then re-compile them. However this can be rather laborious and can be very time consuming.

Presented here is a solution to the problem which can be used while entering and de-bugging a program. It allows a user to redefine a word such that all words that use that word will *automatically* execute the new version. How is this done?

In reality, it is an illusion. The words that use the changed word are *not* in fact changed at all. The original word is actually changed.

ED: and ;ED

Presented here are two very small words (totalling 98 bytes in size) called ED: and ;ED (short for EDIT).

The best way to illustrate their use is by example.

```
: HELLO ." HELLO WORLD" CR ;  
: TEST HELLO ;  
TEST
```

When TEST is called, "HELLO WORLD" is displayed. We will now change the behaviour of HELLO using ED: and ;ED as follows:

```
ED: HELLO 20 0 DO I . LOOP ;ED
```

Now try it by running TEST:

```
TEST  
0 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ok:0
```

As can be seen, we changed the definition of HELLO, but, crucially, we did not have to re-compile TEST in order to see the new behaviour. How is this done?

It is actually an illusion. The replacement code is compiled as normal into memory (less its dictionary entry, because it is not required) and the *original* dictionary entry is *modified* to point to the new code. Thus, it is not necessary to scan and modify all the other words in the dictionary that reference the changed word; as far as they are concerned, nothing has changed.

The code to produce this behaviour is as follows:

ED: and ;ED Code

```
: ED: BL WORD FIND IF
    2+ DUP                \ PFA PFA
    ['] BRANCH           \ PFA PFA 'BRANCH
    SWAP !               \ PFA
    2+                   \ PFA+2
    DUP                  \ PFA+2 PFA+2
    HERE                 \ PFA+2 PFA+2 HERE
    SWAP -               \ PFA+2 OFFSET
    SWAP !               \ --
    ]                    \ SWITCH COMPILER ON
ELSE TRUE ABORT" not found" THEN ;
: ;ED $832E ,           \ COMPILE EXIT INTO NEW DEFINITION
  [COMPILE] [           \ SWITCH OFF COMPILER
; IMMEDIATE
```

In traditional horizontal form, without comments:

```
: ED: BL WORD FIND IF 2+ DUP ['] BRANCH SWAP ! 2+ DUP HERE SWAP
  - SWAP ! ] ELSE TRUE ABORT" not found" THEN ;
: ;ED $832E , [COMPILE] [ ; IMMEDIATE
```

Code Explanation

First, the code looks up the executable address of the word to be edited. It pokes the op-code for `BRANCH` into this address. Next, it calculates the relative address between the `BRANCH` instruction, and `HERE`, where the *new* code will be compiled, and pokes this into the cell after `BRANCH` (this forms the parameter for `BRANCH`).

Then the compiler is switched on. This causes the new code to be compiled to `HERE` as normal.

When `;ED` is encountered it immediately executes since it is an immediate word. It simply compiles the op-code for `EXIT` (832E hex) into the new definition, completing the new definition.

Then the compiler is switched off with `[` (`[` is an immediate word so `[COMPILE]` is used to force it to be compiled into `;ED`).

Acknowledgements

The idea for this technique came from an article written by E.H Schmauch, and was published in Forth Dimensions, issue 3, Volume VI, September/October 1984.

The FIG listing as published contained a drawback, as it actually compiled another `DOCOL` call to the new code. This causes a performance penalty as each re-definition causes an extra level of nesting, so the more times a word is re-defined, the slower it becomes. The version published here does not degrade with each re-definition since it simply compiles a `BRANCH` (requiring no extra nesting).