EuroForth 2025

Forth 2025

Abstract

Forth is stuck in a rut. Much energy seems to be spent in making tiny refinements to a language specification that is 25 years old. In this paper, I shall propose some bold and radical changes, with the intention of returning Forth to its proper place as a useful and modern language.

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

It often feels like Forth is moribund! In the last few years, at this conference, I seem to be in a minority of delegates from commercial companies that use Forth in their main products. Yet, Forth still has huge advantages, and in the last few years at EuroForth, I have tried to highlight extraordinarily useful techniques that are possible in Forth and *are not possible in any other language that I know of.*

One has to ask, why do more people not use Forth, given these advantages? Here are some possible answers:

a) These advantages are not stressed, in online descriptions of Forth. Instead, one sees long descriptions of how antiquated the language is, and how quirky, and by inference how it is only chosen nowadays (rarely) for "niche" applications.

b) If someone, who is assessing which new language to choose, accidentally comes across the Forth standards website, they will ask (after a near death experience due to terminal boredom) "yes, but what is it for, what can it actually do, that others cannot?" - and no answer is given.

c) Forth really does have some antiquated absurdities, the reasons for which are lost in the mists of time.

d) Forth does not initially *appear* to have many features that are essential for modern programming.

2. What are the true advantages of Forth?

a) You can do things *during compilation*. This includes quite complicated things like querying databases. This feature opens up a whole realm of extraordinary possibilities, some of which I have attempted to demonstrate at many previous conferences.

b) In fact, you can do *anything* in Forth. Although there may be guidelines, you are not prevented from doing anything you like. (Of course, this also enables an incautious programmer to get into deep trouble.)

c) Forth can be completely freely formatted. This *enables and encourages* you to write concise, highly readable and easily maintainable code. (Of course, it also enables you to write complete gibberish, if you really want to.)

d) Forth is interactive, and you can easily arrange for this to continue *even while your application is running*.

e) The edit-compile-execute-test-debug cycle is extraordinarily quick and efficient in Forth. This is mainly because everything except the edit is done within Forth itself. The old adage "Forth is its own compiler" is still just as true today.

f) If there's some Forth word you don't like, you can always redefine it.
See section 5 below!

3. What previous advantages of Forth no longer apply?

In the past, Forth was often described as being fast, compact, and so simple that anyone could write their own compiler in a day.

It is true that Forth is still fast and compact. However, to achieve speed when targetting a modern and highly complex CPU, you need an optimising compiler, which is not built in a day.

As regards compactness, who cares any more? If your program won't fit, spend 5 Euros on some more memory.

4. What do we need to do?

- Get rid of the bad bits.
- Enhance the good bits.
- Add the missing bits.

5. The bad bits - WITHIN

I have never known a word, in any language, that has caused so much trouble to so many programmers.
3 1 3 WITHIN BOOL. False ok

The worst thing about it is that because this affects a boundary condition, a mistake may not be seen until an application has been running for weeks.

We got to this state because of incorrect mathematics, which assumed that the input parameters were real numbers. But they are not - they may only be integers. The correct mathematics is to ask whether the integer 3 is within the set {1,2,3}.

I guess it's not possible to completely get rid of WITHIN, but one could at least move it to the "Optional Badwords" wordset! Then we could introduce a new core word - perhaps MEMBER - which is inclusive.

At the very top of every build file, we have to redefine WITHIN. It occurs 1094 times in our main application.

6. The bad bits - CASE

It's the default clause of a CASE construction that causes so many programming errors. Again, the mistake might not be seen for a long time. The perfectly simple solution is to keep the index on the return stack instead of the number stack.

At the very top of every build file, we have to redefine CASE and all its other words. A CASE construction occurs 830 times in our main application.

7. The DO...LOOP conundrum

Observations

a) There are 1612 DO...LOOP constructions in our main application.

b) We currently have a rule that if either of the two input parameters is not a constant, then we always use ?DO instead of DO. We have 1018 ?DOs and 594 DOs.

c) Earlier on in our development process, we defaulted to 0-based indexing (computer friendly). About two years ago, we changed to 1-based indexing (human friendly). We did not modify old 0-based code.

d) We have 1281 instances of either 0 DO or 0 ?DO.
We have 329 instances of either 1 DO or 1 ?DO.
In only two cases out of 1612 do we use anything other than 0 or 1 as the second parameter.

e) A disadvantage of using 1-based indexing is that the first parameter for DO frequently requires 1+. Forgetting to do the 1+ is a common cause of programming errors.

f) There are only 3 instances of +LOOP.

Conclusions

a) Move DO, ?DO, LOOP and +LOOP to an optional wordset.

b) Introduce a new and much simpler looping construct, perhaps
FOR (n---) ... NEXT which loops n times, but skips completely
if n < 1.

c) Retain the same return stack structure as a DO...LOOP, so that
I, J and LEAVE work as before.

This would satisfy 99.8% of our loop requirements, with the benefit of greater security and simplicity.

8. Values not variables

As far back as Euroforth 2000, I started advocating for the deprecation of VARIABLE, @ and !, and the promotion of VALUE. The reasoning for that may be found in my previous paper.

Modifiers (they should not be called operators) should be standardised, enumerated and extendable. Attempting to apply a modifier to an unmodifiable word should give a compilation error.

At present, our main application has 105 remaining VARIABLEs - these are mostly because we have not yet redefined the VFX chain functions, which currently require a VARIABLE root. By contrast, there are 1334 VALUEs.

The issue of thread local VARIABLEs or VALUEs will be discussed in section 13.

9. Add VINDEX, VMATIX, STRINDEX

What modern language could be without proper standardised support for arrays and matrices?

My Euroforth 2000 paper also proposed:
a) VINDEX for an array of CELLs
b) VMATRIX for a two dimensional matrix of CELLs
c) STRINDEX for an array of strings.
d) VFIELD and derivatives, for structures

It turned out that some very large VINDEXs and VMATRIXs were wasting a lot of space by using 64 bit cells when for example only byte values were required. We have therefore extended the concept to provide byte, word and int (32 bit) flavours.

By now, we can't imagine how we managed without them.

There are 210 VINDEXs, 28 VMATRIXs and 26 STRINDEXs in our main application. This excludes the byte, word and int flavours (there are for example 15 VBINDEXs), and the dynamically generated VALUEs, VINDEXs and STRINDEXs resulting from the replacement for the Windows registry settings, also described in a different 2020 paper.

I should add that I am not particularly happy with the naming of some of these words, and would be quite happy to change the names, provided the new name was shorter.

10. Add ENUM<<

How is it possible for a language to survive without a standardised enumeration function? I described a fully Forth faithful enumeration at Euroforth 2023, and some enhancements were described earlier at this conference. Surely, this should at least form a part of an optional word set?

11. Zstrings

It must be at least 30 years ago that I first started advocating a general move from counted strings to zero terminated strings. The original reason was that any serious application is likely to need to interact with the native operating system, and also external libraries with "C" interfaces, and, of course, Windows, Linux and C libraries all use zstrings. Back in the days of Windows 95, it was not certain which side the coin would fall, and the 32 bit Windows version of our main applications extensively used "bi-strings", that is zero terminated counted strings. By the time we moved to Linux, initially 32 bit then soon after 64 bit, it became clear that we were in a zero terminated world, and support for both cstrings and bi-strings was dropped.

As I wrote this, I searched for "zero terminated" in the Forth standard and got "NO RESULTS" - which gives the clear impression that Forth is not a suitable language to use with Windows, Linux or C libraries!

The four essential words needed are:

a) A word to define a z-string - currently:
: Z" (Comp: "ccc<quote>"--- ; Run: ---zaddr)
However, I note that the simple word " is still unused in standard Forth, so why don't we use that?

b) A word to type the z-string (used only for debugging) - currently:
: Z$. (zaddr---)
But see section 18 below.

c) A word for concatenating zstrings - currently:
: Z+ (zaddr1,zaddr2---zaddr3)
Note that this word MUST be thread safe, and must not involve any garbage collection. ***Also see section 18 below.***

d) A word to format a number as a zstring - currently:
: ZFORMAT (n---zaddr)

There are 3446 instances of Z", 1482 instances of Z+ and 432 instances of ZFORMAT in our main application.

Also frequently used, are:
: ZDIGITS (n1---n2) \ If n1 is zero, return zero. Otherwise assume n1 is the address of a zero terminated string, and convert the string to a number.
: Z= (z$1,z$2--- f) \ True if two zstrings are the same
: Z<> (z$1,z$1---f) \ True if two zstrings are not equal
: ZCAT (z$1,z$2---z$1) \ Concatenates z$2 to the end of z$1
(The user responsible for the buffer z$1 in the above)
: ZINITIATOR (zaddr---zaddr') \ Find start of a zero initiated string
: ZTRAILING (z$---) \ A zstring is adjusted to exclude trailing spaces
: ZLIMIT (saddr,maxlen---) \ Adds a zterm to a string of specified maximum length
: ZMOVE (src,dst---) \ According to the documentation, all this does is show off the VFX optimiser!!

In addition, a subset of the file access words are, as needed, converted to use zero terminated names.

12. UTF8

Forth needs to realise that the world has moved on from ASCII. The use of UTF8 is almost universal. Fortunately, UTF8 strings are single zero terminated, so all the zstring words above still work.

I am not quite sure what the XCHAR wordset in the Forth standard is for. All our applications are dynamically multilingual - yet we have never needed any of the XCHAR words.

One thing definitely needs fixing - C@. What this ought to do is fetch the UTF8 character at the address. If byte acting words are really needed, they should be B@ and B!.

13. Threads

These days, only the most trivial applications run in a single thread - yet Forth has no standard wordset for handling threads. Anyone approaching Forth for the first time would be under the impression that Forth does not do threads!

VFX does of course have quite good thread support, but there are three important improvements that are needed.

a) "User" variables (effectively thread-local variables) need to be converted to thread-local values.

b) A method of initialising the thread-local values is needed.

c) A better method of dealing with thread-local memory is needed.

There are 40 threads in our main application, though typically only half a dozen are running at the "same time".

14. Execution chains

This has been a feature of MPE Forths for many years, but it is only recently that we have discovered how extremely useful it is. This is another example of a concept that is easy in Forth, but quite hard if not impossible in most other languages.

15. Libraries and externs

Yet again, we have a situation that the Forth standard does not mention something that is a necessity for any serious application!

16. Databases

And again, how many real life applications need to access databases? The Forth Query Language (FQL) has been around for years and is rock solid. Dare I have the impertinence to suggest it should go into the standard?

17. Locals

A frequent criticism of Forth is that stack manipulation makes it hard to read. Stack manipulation should be de-emphasised in favour of locals.

The Forth standard needs to be completely rewritten to use:
{ <ins> | <locals> -- <outs> }
where standard ins and locals behave like VALUE (with all modifiers permitted), and other data types (e.g. float, string) are available. Locals should be automatically initialised.

17. Doubles

Think of one of the most popular and inexpensive tiny computers - the Raspberry Pi. It has a 64 bit CPU. Do we really need double length integers any more?

Let us try and think of some very large numbers, for example the US national debt. At the instant of writing, this stood at $37,289,586,478,935. The largest unsigned number representable in 64 bits is 18,446,744,073,709,551,615. We could express the US national debt in cents and still have plenty of headroom!

The reason for raising this issue is because when we demonstrate interactive Forth to a newcomer, they quickly understand it.

2 2 + . 4 ok Nice!

But shortly afterwards, they will try:

1.2 3 + . 3 ok-1 WHAT???
That takes a lot of explaining, and it is completely unnecessary.

18. Numbers, and things

This brings me to possibly the most radical proposal, which is to address the issue of why 1.2 3 + does not work, when it so easily could work.

The first and easiest fix is that when we are free from double numbers, the floating point recogniser could accept 1.2.

The next thing to think about is data types. From the birth of Forth, there was an assumption that *everything* was integer. There was no need to consider data types, because there was only one.

We were told that floating point was slow, used a lot of expensive memory, and was unnecessary.

This has not been true for decades. Floating point is just as quick as integer, uses the same amount of memory, and is essential. So a floating point wordset was tagged onto Forth, and you had to remember to put an F in front of everything.

Furthermore, type conversion is manual, and you have to remember where you are on two different stacks.

There are 544 instances of S>F and 144 instances of F>S in our main application. There are no instances of F>D or D>F. This leads us to consider that maybe in the future floats should be the default, and that to specify an integer you need to put a prefix, just like you do to specify a hex number.

A more interesting idea might be to merge the data stack with the floating point stack, and instead add a type stack. The basic addition word, and many others, could then become type smart. Now we are away!

```
1.2 3 + . 4.2 ok
" abc" " 123" + . abc123 ok
3 1 3 WITHIN . True ok \ YEA!!
" 欧洲会议" 2 + . 会议  ok  \ That is "European conference", by the way.
```

Against that, there is sure to be the argument of speed. However:

a) Nowadays, most computing time is spent inside library calls, not in Forth itself.
b) If your program runs too slowly, spend 5 Euros on a faster PC - it's much cheaper than buying a programmer's time.
c) Constant arithmetic of mixed type as in the examples above would in any case be resolved by the optimiser at compile time, not at run time.
d) If you really, really needed to speed up some critical routine, the explicit arithmetic and type conversion words would still be available.

19. Conclusion

I hope that these notes and observations will lead to some radical changes to the Forth standard very quickly, so as to restore it to being a language of choice for discerning programmers.