w3forth - make IT small again

- FORTH für non-FORTHer (German)
- The power of indirect threading code (ITC)
- Implementing a FORTH virtual machine
- FORTH: getting used with data stack
- My answered QUORA questions mostly targeting FORTH
- STABLE an extreme minimal and fast FORTH-VM

FORTH and the art of simplicity

One thing seems to me is gone the last couples of years - simplicity.

There are some programming languages out there which enforces purity in some concepts. For example SMALLTALK for object orientation, HASKELL for functional programming.

I would say FORTH enforces simplicity. To do simplicity FORTH is not a requirement, like Smalltalk is not a requirement for writing object oriented applications. FORTH is an eye opener in this respect.

Before I'm going deeper into detail I would enumerate some fundamental concepts within FORTH.

* FORTH is very small, about some k of size. This mean that there is not much flesh which have to be maintained and understand.

* FORTH doesn't rely on any operating system but works nicely on top of most operating systems.

* FORTH is a realtime operating system.

* FORTH contains an interpreter, compiler, editor, debugger, persistence and multitasker. For an experienced FORTH user this is builds a remarkable productive system.

* FORTH is self contained, that means that the interpreter, compiler, debugger, etc. are always available, even at runtime and still requires only some k of memory. It is convenient to save the current system as a new executable and start from there or distribute that executable. No other programs are required (no external compiler or linker).

Using and programming FORTH is brutally simple, but it seems to take years to master it. Since FORTH is a stack oriented language most developers seems to have troubles in switching the kind of thinking. A big source of problems arises if developers try to apply their known features of their current tools to FORTH. FORTH is working in a very different way to be able to achieve simplicity.

I don't show FORTH code for now, because it is like a foreign spoken language. It will merely distract the user. So I continue to elaborate the principle behind the simplicity first.

FORTH is about problem solving. Real problems, not self made problems. It seems to be difficult to see self made problems, so it is important to focus on the customers requirement and to ask about the source of the problem which is going to be solved. For example configuration files, the need for garbage collector, SQL database, Frameworks and so on.

In writing libraries and frameworks without knowing the problem will end in a large and less flexible systems which doesn't meet (perfectly) the requirements. SQL databases are a good example. Because we have to work with SQL we squeeze the problem into tables, fields and relations. This approach works well, but is far away from optimal, in my experience about three order of magnitudes. Therefore I gave up on SQL and switched to a direct memory mapped storage system.

In reducing complexity we get safety. Another example are files. Files are not a big thing, but I/O can raise many exceptions (disk full, read or write errors, file not found, no open-read-write permission etc.). In avoiding files we have the opportunity to remove 100% of that complexity. Files are a requirement, how could we abandoned files then ?
One approach is to factor out the reading and writing of files in a different, independent tiny application. This application transform the files into a reliable alternative (i.e. fixed size memory map file or any other idea). Or vice versa, from, for example, memory into files.
But these converter are not part of the business application.

This raises the idea of independent tiny applications. Each application solves an independent part of the whole application. With this approach the need of namespaces disappear. The need of dynamic memory management disappears.
At the end of the tiny application all memory will be reseted, a 2 nano second operation. It should be easy to prove and test tiny applications. Changes in one tiny application don't interfere with other tiny applications.

But how can we get there ? First we need a deeper understanding of the requirements and the solution. Writing small prototypes might help here. Split the requirement into independent parts. UseCase analysis might is a good approach. Then only implement functionality which is really needed.

05.02.2018, Andreas Klimas - klimas@w3group.de