Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Modern Forth

Modern Forth systems are not like their ancestors

September 11, 2008
URL:http://www.drdobbs.com/embedded-systems/modern-forth/210600604

Stephen Pelc is managing director of MicroProcessor Engineering Ltd. He can be contacted at www.mpeforth.com

---

> Engineering is the art of making what you want from things you can get. **--Jerry Avins**
>
> I am a design chauvinist. I believe that good design is magical and not to be lightly tinkered with. The difference between a great design and a lousy one is in the meshing of the thousand details that either fit or don't, and the spirit of the passionate intellect that has tied them together, or tried. That's why programming (or buying software) on the basis of "lists of features" is a doomed and misguided effort. The features can be thrown together, as in a garbage can, or carefully laid together and interwoven in elegant unification, as in APL, or the Forth language, or the game of chess. **--Ted Nelson**
>
> We know about as much about software quality problems as they knew about the Black Plague in the 1600s. We've seen the victims' agonies and helped burn the corpses. We don't know what causes it; we don't really know if there is only one disease. We just suffer -- and keep pouring our sewage into our water supply. **--Tom Van Vleck**

There is increasing realisation that there are no magic bullets for software development. Interactive languages returned to fashion in the web developer world, which also saw a return to multi-language programming -- use each language for what it's good at. I'm going to look at one of these interactive languages, Forth, and show what people are doing with it, and why it's a powerful tool.

Modern Forth systems are not like their ancestors. The introduction of the ANS/ISO Forth standard in 1994 separated implementation from the language. One result was a proliferation of native-code compiling systems that preserved the traditional Forth interactivity while providing the performance of batch-compiled languages. It is the combination of performance and interactivity that makes modern Forth systems so attractive.

What is Forth?

Forth is a member of the class of interactive extensible languages. Interactive means that you can type a command at any time and it will be executed. Extensible means that there's no difference between functions, procedures and subroutines (called "words" in Forth parlance) that you defined and the ones that the system provides. Forth is an untyped language whose primary data item is a cell. A cell is the size of an item on the stacks, normally 16, 32, or 64 bits.

Underlying all languages, there's an execution model or virtual machine (VM). C and the other languages have one and so does Forth. Forth is rare among languages in exposing the VM to the programmer. The VM consists of a CPU, two stacks, and main memory; see Figure 1. The stacks are conceptually not part of main memory. When I refer to "the stack" I mean the data stack.

[Click image to view at full size]

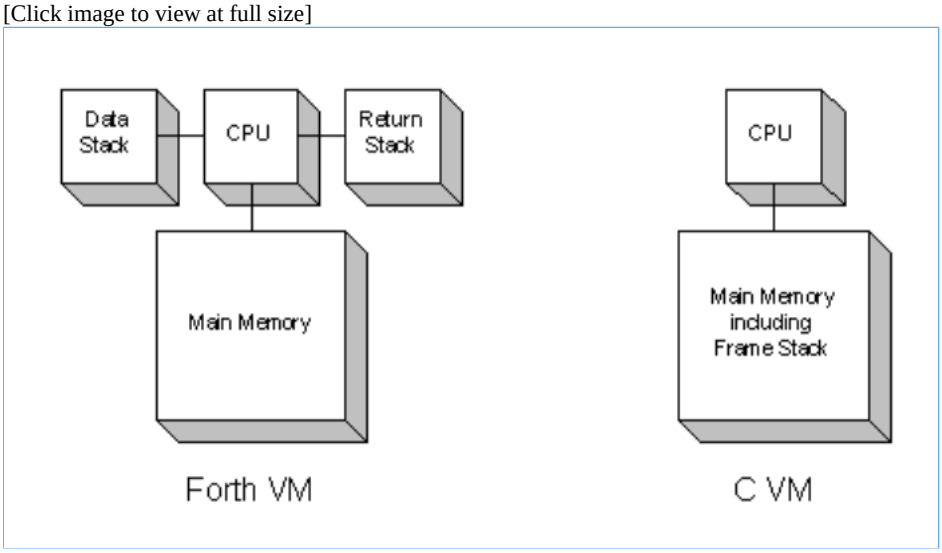

Figure 1: C and Forth VMs

The two stacks are called the data stack and the return stack. The data stack holds arguments to words, return values, and transient data. The return stack holds return addresses and can also be used for temporary storage. There are several consequences of having two stacks, the most important of which are:

- return addresses do not get in the way of the data stack, so words can return any number of results.
- Items on the stack do not have names -- they are anonymous.

Because items on the stack are anonymous, you sometimes have indulge in stack juggling using words like **DUP** and **SWAP** to manipulate the stack. In the spirit of the marketing principle "advertise your worst feature", the anonymous stack leads, in part, to Forth's well-deserved reputation for small applications. Because stack juggling requires thought, you improve productivity by reducing thought. Many small words reduce the thought required by splitting the job into manageable chunks. A side effect of this is that you reuse code at a much finer grain in Forth culture than in most others. The ease of code reuse at this level leads to small applications. Factoring at this level means that access to many performance-critcal routines is through small well-defined interfaces. A consequence is that you can make major changes easily. For example, an embedded system can move from EEPROM data storage to a file system very quickly. Forth is a very agile language.

For those who are unfamiliar with Forth, the Forth text interpreter is a little odd. It deals with whitespace delimited tokens. These tokens are looked up in a dictionary. If found they are either executed or compiled. If a token is not found, its is considered to be a number. If number conversion fails, an error occurs. That's all! A consequence of this is that a valid Forth word name can contain any non-whitespace characters. Because there's a lot of interactive testing, many commonly used words are short. The common strange-looking ones are **@** (fetch), **!** (store) and **.** (integer print). The collection of word names is called the "dictionary," which can be split into vocabularies (namespaces). For more about the Forth interpreter, see JForth: Implementing Forth in Java by Craig A. Lindley.

Some modern Forth implementations include:

- MPE Forth
- SwiftForth
- iForth32

I don't have the space to teach you Forth, and I've referenced a few tutorials at the end of this article. I have also referenced a few modern Forths that can be freely downloaded.

Performance and Interactivity

Modern Forths do not generate threaded code any more, except for very particular sets of circumstances. Modern Forths generate native code using the same techniques as compilers for other languages. When we implemented our VFX code generator 10 years ago, we found that there is only one algorithm needed specially for Forth, and even that has an analogue in some C compilers. Consequently the limits to Forth performance are the same as for any other language -- design objectives, manpower, money, and time.

The requirements of interactivity do introduce some constraints. Forth programmers expect their compilers to be fast. A client's application containing 80,347 words in 815,548 lines of source produces 15MB of binary in under a minute on an old 2.8GHz P4 box. Recompiling the application reduced compilation time from 56 seconds to 34 seconds, showing the impact of hard disk performance. Even on this old PC, we're seeing compilation speeds of nearly 25,000 lines per second generating nearly 500KB per second of binary. For an embedded application it is common to see compilation times under one second, and for small applications less than 0.1 seconds.

There are several consequences that affect the way Forth programmers can work. Compilation time is unimportant compared to testing time, which in turn means that changes in low level code can be investigated very quickly. We'll come back to this later. In most modern Forths, cross-reference and source location tools are built-in, and the compiler and symbols remain active during testing. Forth has a strong focus on debugging.

Interactivity and Debugging

We know from experience that the productivity of programmers varies over a huge range. A large part of this has nothing to do with coding, and a great deal to do with project management and the ability to debug. There's nothing magic about debugging, it's just the application of formal scientific method:

- Observation
- Hypothesis
- Experiment with a yes/no answer
- Conclusion

[Click image to view at full size]

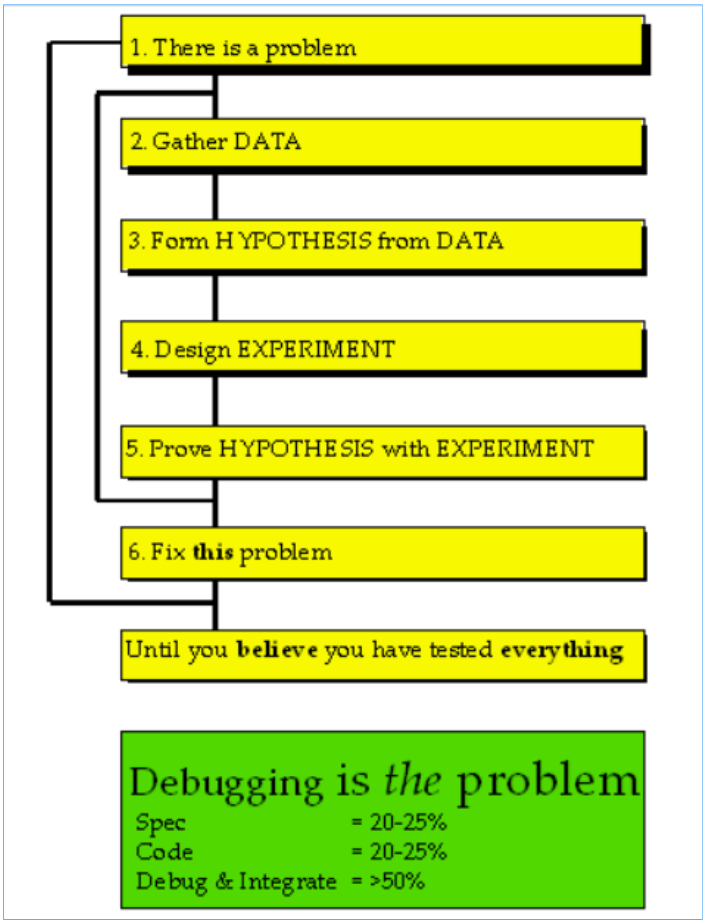

Figure: 2 Debug

Figure 2 shows that debugging consists of two nested loops. How fast you can go around the inner loop determines how fast you can debug a system. Interactive debugging is the fastest route I have found. The stages of debugging are:

1. Make the problem repeatable. This usually involves finding out which inputs cause the problem.
2. Gather data about the problem. Observation is crucial, so take this stage slowly and carefully. I have seen people immediately dismiss exception displays and crash-dumps which contain vital clues.
3. From the data, form a hypothesis as to what caused the problem.
4. Design an experiment which tests the hypothesis. The important part in designing the experiment is to ensure that it gives a yes/no answer.
5. Run the experiment. If the hypothesis is incorrect, go back to stage 2.
6. Fix the problem.
7. If you have more bugs to fix, go back to stage 1 for the next problem.

Programmers are very rarely taught how to debug. The problems they face are almost always in their source code. So why don't they read it more carefully? In particular, why don't they write a line or two describing the whats and whys of of a routine before they write the code? The poor relation who maintains your code for the next 10 years costs your employer a fortune because you could not be bothered. As a sidenote to this potential rant, my company has been using literate programming for 10 years and we believe that it has increased the quality of our code. Whenever we incorporate third-pary code into our code bases we document it to our house standards. This process nearly always reveals bugs. To produce stable software the golden rule is to fix the bugs first. A second rule is not to write clever code. Although this upsets people trying to achieve alpha-male guru status, you should remember the aphorism attibuted to Brian Kernighan:

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

Forth is a wonderful tool for debugging systems and their software. Since we spend most of our time debugging and testing, it astounds me how our industry accepts mediocre tools for critical activities.

When I'm programming embedded systems that control mechanisms, I always treat the hardware specifications with as much concern as the software specification, which probably doesn't exist. A large number of things you have no control over are changing around you on a daily basis; gear ratios, airline lengths, and more. The team proudly put the prototype bomb-disposal machine together according to the ministry specifications. Then the joint services bomb-disposal team came to visit. No bs, just "No, we don't do it that way, we want it done this way". The conversation with the navy people about doing this underwater was memorable. We all admired that team for their useful directness as well as for the job they do.

Of course, the management just said we could fix it in the software and we can't change the delivery deadline. This is known in our shop as "tail-end Charlie syndrome". The software team accumulates all the upstream problems, the mechanics are delivered late, the air lines are of a length to cause resonance problems at certain motor speeds, and the control system needs a rewrite when the four-axis frame is at the bottom of a muddy trench being sprayed with abrasive grit. The fourth air motor driver was acceptable (Figure 3).

[Click image to view at full size]

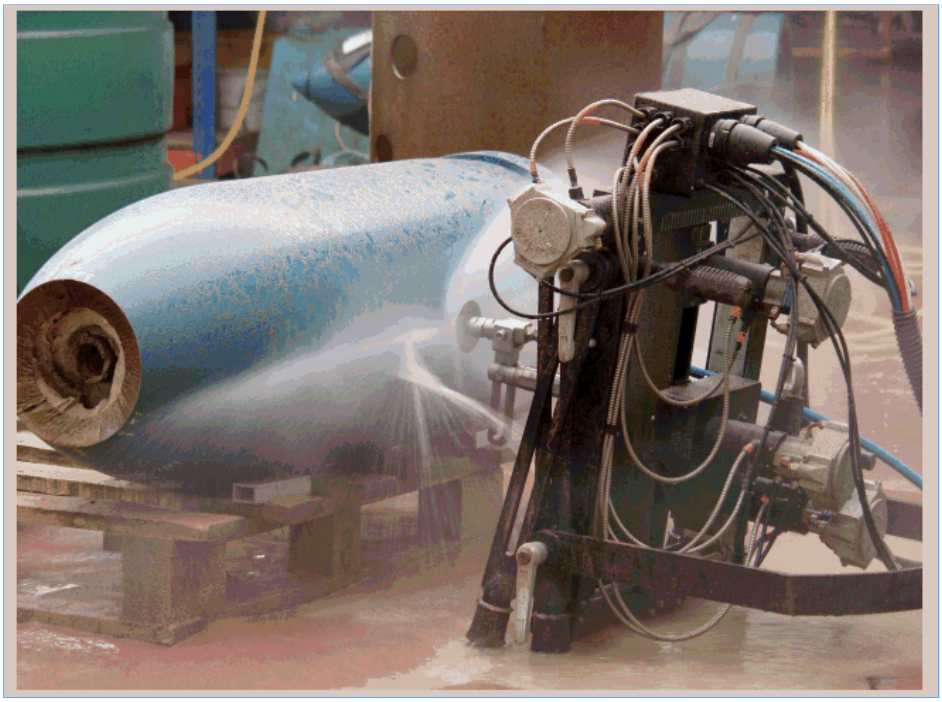

Figure 3: RE 220 ACE in ideal conditions

Commissioning takes far longer than anyone likes to admit unless it has been properly planned for. I'm old enough to have seen a few projects go wrong. Fred Brooks said "Good judgement comes from experience, and experience comes from bad judgement". In the majority of cases software projects go wrong for lack of good project manangement. It is part of your resposibility as a software engineer to train management to understand your problems and to plan for them.

During development, you will not predict the problems you will actually encounter during the product life, so you cannot provide the test code for those problems. A remote console with full access to all words becomes a lifesaver. The bomb-disposal machine contained two embedded systems and a Windows panel PC, all running Forth systems linked by an RS485 network. The Forth interpreter in each system was available at all times. Some of the printable stories from this project are available in the EuroForth 2003 conference proceedings.

Document As You Go

Even in the middle of the bomb job's 100-hour weeks, I insisted that we maintain our house coding standards and documentation. There are three reasons for this:

- It reduces bugs early on. A rule of thumb indicates that a bug costs three times as much to fix at each release stage after writing. Early documentation of each word/function will reduce your bug level regardless of the programming language. This alone pays for the effort and reduces delivery time. Make sure that you document why as well as what. Your maintenance programmers will thank you.
- Managers like manuals. They may not understand them, but a neatly formatted PDF manual impresses and gives a good indication of programmer mindset and whether this code will be maintainable.
- Especially if your code is safety critical, verification, and validation of some form will be required. Doing it after the code has been written is an awful job.

We use a literate programming system called DocGen. Doxygen and friends are available for many other languages. The cardinal requirement for any such system is that the documentation is never separated from the code. If it is, the documentation and code will become out of step as soon as you enter thrash mode.

A good literate programming system will allow you to generate chapters, sections and simple formatting. If you can include images and generate an index, so much the better. Other facilities that can be useful are a selection of output formats, e.g. PDF and HTML, and conditional generation which allows you to produce different versions for programmers and end-users, or to produce different versions for different operating systems.

Applications

To use a language well, you have to exploit its best features. With an extensible interactive language such as Forth, the interpreter and compiler can be available a runtime. You can use them for configuration scripts, application macros, CGI scripting, SOAP servers, XML parsing and remote debug access. End-users can use the tools too if you provide a set of words tailored to their needs. You can treat Forth as a toolkit for producing application-specific languages. The two examples below come from DOS and Linux systems:

```
[COM  COM2: 38400 baud  N,8,1  COM]
```

which defines how COM2 will be used.

```
  s" /dev/ttyS0  115200 baud  8n1  1 dtr 1 rts DOS"
  r/w sd0 open-gio ...
```

which opens a Linux serial port as required for a building management system. The text after the device specification is interpreted by the word **open-gio**.

In hosted system, we use mini-languages for parsing resource scripts and for defining external shared library interfaces. The following code fragment is a Forth definition.

```
Extern: BOOL PASCAL PostMessage(
  HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam
);
```

The three previous examples take advantage of Forth's extensibility. The compiler itself is available to Forth programmers, who use it to produce new compilers. The word **Extern:** above produces both a Forth word and a code generator.

The use of interactivity enables remote debugging across the world. Our PowerNet TCP/IP stack provides a Telnet server. You can access the remote Forth system's text interpreter to find out what is going on. Although we tend to think of this as big system functionality, it's a reality in single chip systems these days. A build of a an open Forth system (interpreter and compiler) with serial and Ethernet drivers, TCP/IP stack, DHCP and multi-threaded web and Telnet servers requires about 116KB of code space and 32KC of RAM (we've also done it with 16KB of RAM). Single-chip 32 bit microcontrollers provide enough code and RAM space.

We needed the interaction when a client's client-placed instruments on a raw Internet feed with no protection at all. The instruments were attacked within four minutes of being connected. After investigation using a Telnet connection from 4000 miles away, we were able to see the symptoms, diagnose the problem and enhance PowerNet's security to pass Linux server tests. The on-board Forth and connectivity saved time and money.

The Candy system is a project control package for the construction industry by Construction Computer Software (CCS); see Figure 4. It is used all over the world for large and small building projects. It planned the Hong Kong airport and metro. Candy consists of about 850,000 lines of Forth source code built on VFX Forth for Windows. The core development team is remarkably small and their productivity is dependent on strong project management using a surgical team approach. The application layer team uses an object oriented approach and notation designed for people who know about construction. Interactive debugging is available in the running application.

[Click image to view at full size]

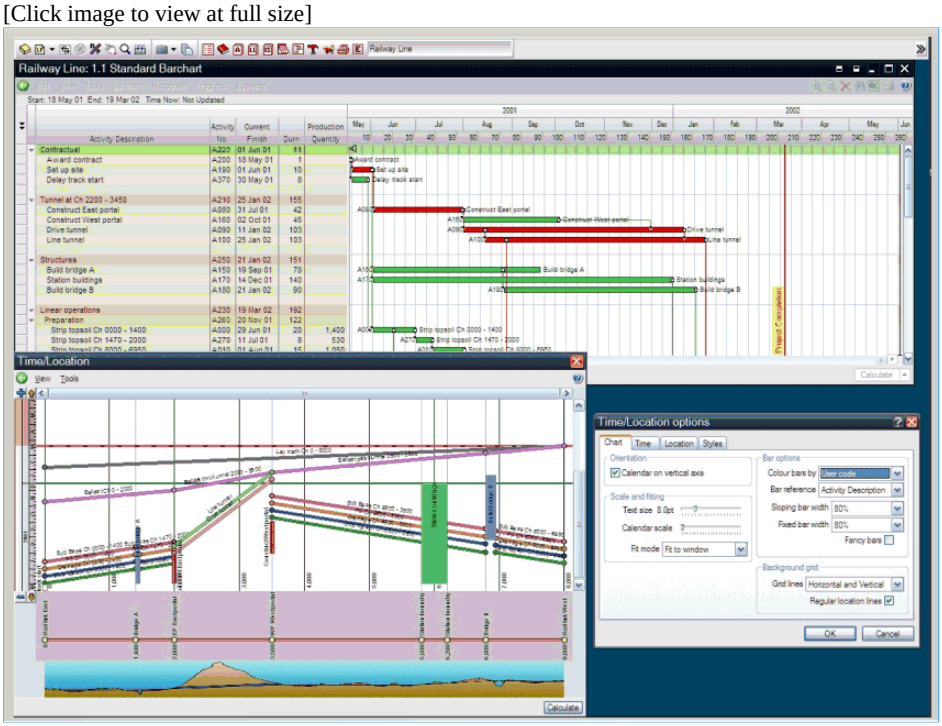

Figure 4: CandyCCS

Commercial laundries use gravity driven monorails, points (switches in North America) and lifts to move bags of washing around. The control system and panel for one of these is Tracknet, a Forth application by Micross Automation.

[Click image to view at full size]

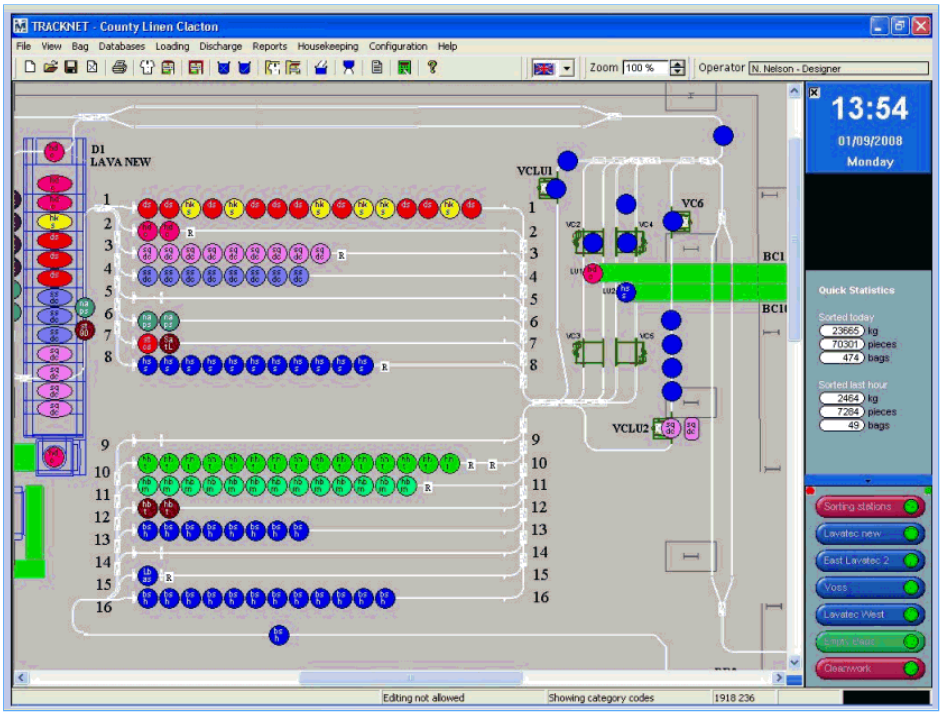

Figure 5: Tracknet

What's Next?

In a future article, I describe some of the experimental and bleeding edge Forth systems, including changes to the Forth VM and multi-core stack machines in silicon. You thought your four core box was hard to program -- wait until you have 24 or more cores to deal with.

Acknowledegments

- Construction Computer Software, Cape Town, South Africa
- MicroProcessor Engineering, Southampton, UK
- Micross Automation, Ross-on-Wye, UK
- Richmond Engineering, North Lopham, UK