Thoughtful Programming
Chapter 2
The Second 10x

The first Forth in hardware was the Novix. Chuck wrote **cmForth** for the first Forth machine. cmForth was only a few K of object code and about 30k of source. It included a metacompiler to compile itself. It included an optimizing native code compiler for the Novix that was very small and very powerful. It was so fast that when running off a floppy disk it could boot and metacompile itself off the floppy disk at boot before the disk could get up to full working speed.

This phase has been called hardware Forth since the Forth that Chuck wrote was for the Novix which was a Forth in hardware chip. Other people looked at all the innovations that went into cmForth and wrote systems to experiment with some of the things that Chuck tried in that implementation. Pygmy Forth is an example of a system on the PC that was modeled after cmForth.

Chuck was not happy with cmForth however because it was too much a hardware Forth. The Novix had some powerful features but writing a compiler to take advantages of the specific features of this quirky hardware was too complicated and too hardware specific for this chip only. As I say some of the innovations, some of the simplifications in Forth that were implemented in cmForth were ported to Pygmy on the PC and there are groups of programmers who used cmForth and loved it and there are still people using Pygmy Forth on the PC.

Forth had been a very portable language because it was based on implementing the Forth virtual machine on some hardware. Once you did that you were dealing with more or less the same virtual machine on any Forth. Chuck looked at the concept and decided to design an improved Forth virtual machine and design both his new chips and his new Forth for these chips for this virtual machine. He wanted to get away from a hardware specific approach to Forth and go back to using a portable approach to his new Forth.

The result was the virtual machine model that became the hardware implementation on his MISC machines. At first he referred to the native code for his chips as assembler like everyone else but after he wrote a tiny portable native code compiler as the assembler for this machine he decided it should be called Machine Forth. This new Machine Forth was not only smaller than cmForth but the approach was portable again because the simple optimizing compiler could be implemented on any machine by simply implementing the virtual machine just like in the old days. This was the technique Chuck used on his remarkable OKAD VLSI CAD program.

While much of the Forth community was beginning to embrace optimizing native code Forth compilers in their own environments these Forths were very different. Other peoples optimizing compilers could be very complex. I implemented one that had many rules for inlining code and well over one hundred words that were inlined. This is very different than the optimizing compiler that Chuck designed which just inlines a dozen or so Forth primitives and compiles everything else as calls or jumps. It was the smallest and simplest Forth Chuck had built yet. It was not just a very effective programming tool for him but has been used by dozens of other people over the years many of whom just loved it and reported amazingly good results. There were also a lot of people who put their hands over their eyes when Chuck showed people what he was doing.

All of my work for years has been either in ANS Forth or in Machine Forth. There are a number of other people who did a lot of work in Machine Forth and there were different styles employed by different people. One of my jobs at iTV was training new people and exposing them to our chips, our tools, our Forth and our methods for using it. I was pleased to report taking users from no prior knowledge of Forth, our chips or our tools to being productive programmers after a two hour tutorial and having them showing the first of their application code for their project the next day. I like to ask people how long it takes to train someone to be an assembly programmer on the Pentium or even a 'C' programmer just to see if anyone will ever say two hours from scratch.

Using smaller, faster, easier to understand tools and techniques in your Forth can get you some of those factors that can get you that second 10x. Machine Forth source and the technique it uses for compiling is easy to read and understand. Chuck has discussed his reasons for introducing the innovations that he added to Machine Forth and why they produce more efficient code than the original Forth virtual machine model.

Like the first 10x there is not one factor that is 10x. 10x is the result of combining a number of smaller factors. Chuck has explained the first set of these things many times. Mostly they are unpopular because they do not map well to what you have to do once you have gone way done the complexity path. Once you have introduced a large overhead of fat to deal with the lowfat approach doesn't apply well.

Forth is stacks, words, and blocks; start there. Stay on that path, keep things simple, as simple as possible to still meet the other goals mentioned above. If you start to head off that path you start to introduce complexity that isn't needed and can eventually become the biggest part of the problem. Keep the problem small. Focus on solving the problem.

Stacks are great (for Forth) so use them. Chuck has said "Don't use locals." You have a stack for data, this is Forth, locals are for a whole different picture. Words are great (for Forth) so use them. You have this wonderful mechanism for encapsulating code and data, for factoring and data hiding. If you want this stuff it has been there for twenty years, you don't need to extend Forth to look like some other OO language. We already have words. Most other languages implement OO by using late binding while Forth prefers early binding. Why do things at runtime that you can do at compile time? But most OO Forth implementations use late binding and push calculations from compile time into runtime and thus produce overhead.

Blocks are great (for Forth) so use them. I don't say never use files. That would be ridiculous. But in most cases Forth can get a big advantage in small clear source and simple fast object code _where it counts_ by using Blocks where appropriate. Twenty years ago I learned that by substituing BLOCKS for files I could get parts of my applications to go 100 times faster and be easier to write and maintain. I also learned that by using a ramdisk or file caching in memory I could get a 100 times speedup. I experimented with combining the two and using ramBLOCKs on my expaned memory devices. I could get the both the BLOCK and ram speedups and tailor it to the performance balance that I wanted. Chuck has done something similar. Chuck uses BLOCKS in memory as a form of memory managment. Chuck's BLOCK references compile into a memory reference. He has a couple of hundred bytes of code to transfer blocks to or from disk to memory once in a while.

The approach of using only the layers of abstraction that you need can include files but it may be better to have your own file system that meets the needs of an application. I know from experience that a file system can be very small and matched tightly to your hardware and provide things like file caching about as efficiently as it can be implmented. Chuck would say that if you are going to use files that you should do it where appropriate in the most appropriate way. He would say that many people no longer consider BLOCKS.

Common practice has been to abandon BLOCKs altogether. Keep as much of that original 10x as you can. Don't start by giving up the advantages that you had to begin with. Don't use files for everything! Don't use files inappropriately.

Forth provides unique access to a combination of interpretation, compilation and execution that just isn't there in other languages. It is part of the power of the language and it goes well beyond interactive development. It comes from a very very important notion in Forth, don't do what you can do at compile time at runtime and don't do what you can at design time at compile time. Chuck will start with a program that other people will code as A B C D E F G and rethink to find a way to do the equivalent as A B E F G by figuring out how NOT to compile part of the program and still make it work. Then he will figure out how to make B and F happen at compile time by switching between interpret and compile mode while compiling. Chuck's compiled code will be A E G. with C D eliminated (done at design time) and B F done at compile time. Chuck has said that one of the problems he sees with many mega-Forth systems is that they don't switch between compile and interpret modes often enough and compile too much.

On the Forth systems I have been working on for several years I can do a Forth **DUP** in one of three ways. I can write **DUP** in Machine Forth and compile a 2ns 5 bit opcode. I can switch to ANS Forth and write **DUP** in a definition. It will compile the ANS word **DUP**. In most of the ANS Forths this will be about 50 times slower than the Machine Forth but at least it still compiling at compile time and executing at runtime. I can also do it with late binding as is shown in many examples in c.l.f **S" DUP" EVALUATE** This involves interpreting a string, searching the dictionary, eventually executing the DUP with late binding all at runtime. This is a powerful technique that can certainly make Forth look like a completely different language. It is also a good way to slow down the program by a factor of about 600,000 times. I know, who cares? Just buy a new computer that is 600,000 times faster to make up the difference. Is this what we want to show beginners in c.l.f to expose them to good ANS Forth practices? Give them a page of rules for spelling, a page of rules for laying out the code and examples of how to slow things down by 100,000x. This is pushing things from design time to compile time and from compile time to runtime in a big way. The idea in Forth to do just the opposite.

Chuck likes to say that executing compiled code requires that the code be read, compiled, and executed. You have to read it to compile it and then execute the compiled code anyway so consider reading it when you want to do it by using interpretation. If you are dealing with small simple source this seems to make sense. I doubt if it seems to make sense if you have introduced so much complexity that you have megabytes of code. In Chuck's software with a 1K Forth and typically 1K applications interpreting source is fast and involves searching short dictionaries. Interpretation becomes less attractive when it involves searching wordlists of tens of thousands of words. Chuck refers to his concept as **ICE**, Interpret, Compile, Execute.

As a metric I did some analysis of code examples Chuck has provided. The numbers I find most interesting is that **the length of the average colon definition is 44 characters. The length of the longest colon definition was 70 characters.** This is a sign that he has factored, factored, factored. How big are the average definitions in your code? Smaller definitions are easier to code, easier to test, etc. **The ratio of number of interpreted to compiled mode words in his source is 1.65/1. That is he as 1.65 times a much being interpreted while compiling as compiled while compiling.** Chuck says that without this use of **ICE** programs can easily become too large. Finally although Chuck includes almost no documentation in the source code itself it was accompanied by lots of external documentation. **The ratio of documentation text to source code text was 339/1.** Chuck feels a description of the code and explanations can be linked to the code but should not clutter it. He feels that documentation deserves a place of its own and has created various web pages so that the documentation can be accessible to other people.

He says he can take a small Forth and interpret an application from source faster than a megaForth can execute compiled code. He is not just talking about a small difference either. The megaForth may take quite a long time loading into memory, the compiled code may also have to come in as a DLL. I doubt if you can load and execute a few megabytes of compiled code this way as quickly or easily as simply interpreting a a small source file in a system that can load into memory, and compile or interpret applications before your finger can come up off a key.

Chuck would say if you know how to write good code you won't be so reliant on source code libraries. He would say, "**Don't use libraries inappropriately.**" Don't compile everything including the kitchen sink just because it is there. If you don't need it leave it out. Leave it in the library. I have often seen people take a big computer and start coding with a few library includes to get started and then run out of resources before they can get beyond the sign-on message, megabyte hello-world programs.

Use efficient control flow structures. Don't over use them. Saying **X X X X X** is cleaner, simpler, faster and clearer than **5 0 DO X LOOP**. We have so many choices in ANS Forth that we can make it look like anything, but don't. Keep it simple. There are words like **CASE** that Chuck refers to as abominations, keep it simple.

Speaking of abominations, don't **PICK** at your code. Don't **PICK**, don't **ROLL**. Don't delay doing things and never **POSTPONE**. (Compiler writers excepted.) Stay away from the **DEPTH**. Programs that have lost track of what is on the stack are already in big trouble. Chuck has made himself very clear about this and explained his reasoning. **There is a long list of words in the ANS Forth standard that Chuck would say you should avoid if not simply remove from your system.** The problem of course is that if you have ANS Forth libraries they too are full of them. But this just means that you have the opportunity to do it better this time.

One of Chuck's principles is that what you take away is more important than what you add. You have to add whatever you need. The problem is that you end up adding a lot of stuff that you didn't intend to add originally but the complexity just demanded it. The more you add the more the complexity demands that you add more and it can easily get out of control. You have to put in what you need but unless you make an effort to take out what you don't fat accumulates.

So a review of what I would call Chuck's second 10x are first stay with stacks, words, blocks as a start and don't be in a hurry to abandon them for things from other languages like locals, objects and files. Think through the problem, spend your time in the design phase figuring out how to balance the program between interpretation and compiled code execution. Don't just jump in writing code and compiling everything. Use the ICE concept. Don't abuse libraries. Compile what you need not every library and extension. Use simple control flow and layout of the code to enhance the factoring factoring factoring. Use **BEGIN**s rather than **+LOOP** etc. Don't use the long list of unfortunate words that have crept into and been quasi legitimized by ANS. Do all this and don't give up the 10x that you stared with. Keep it simple.

I had several years to observe a team of programmers some of whom were following these guidelines and some of whom were following common ANS practices. Chuck also observed these same practices and solidified his opinion of ANS Forth use. My position as programming manager was somewhat different than his but we tended to agree on what we should and could do.

Chuck has said this stuff many times over the years. If you introduced the word **FOO** into the ANS standard you don't want to hear Chuck telling people not to use it. If you charge people for fixing their buggy programs you may not want them able to write solid efficient programs on their own. If you are promoting your megaForth system you may not like Chuck advocating the use of small and sharp Forths. If your thing is locals or **OOF** or Forth in 'C' or Java then you may not like Chuck's opinions on these subjects. With so many people pre-prejudiced against Chuck's ideas on these subjects there are a lot of people who will trivialize and distort Chuck's position on these subjects out of self-defense for their own position. At the same time some of them have said that they will refuse to even look at Chuck's examples or proofs, they have closed their eyes and minds on these subjects.

Chapter 3

Chapter 1