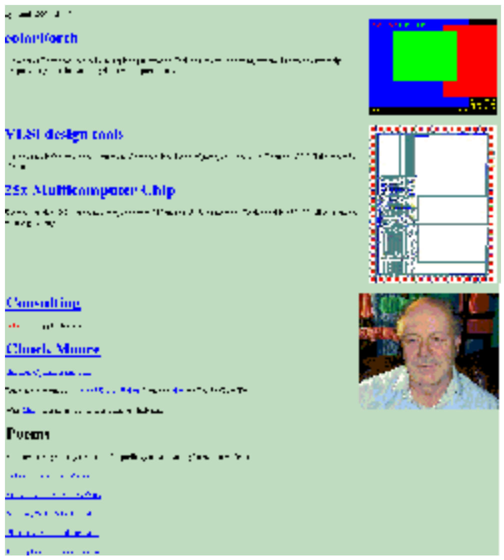
www.**UltraTechnology**.com

(Visit Chuck Moore's Website ColorForth.com to get the inventor's thoughts on Forth.)



Thoughtful Programming and Forth

by Jeff Fox

Preface

Computers are amazing machines. They have an incredible potential. At their best they provide an extension to our minds and our traditional media assisting us in storing and retrieving data, making calculations and decisions, visualizing information, and communicating information between humans. At their worse they introduce nightmarish complexity and problems into our lives. They account for an entire industry that is vast and pervasive and which works in cooperation with strong media and socio-economic forces to sell and promote computer use in our culture.

The technological fads and media hyped product feeding frenzies that we know of as the modern computer industry also have a dark side. The phenomenon known as the digital divide is the way that technology is creating a strong social economic division in our culture that could influence generations. Those with access to modern computers will have access to nearly unlimited information and a world of training, experience and opportunity that the have-nots will never know. The strong and disturbing evidence is that home computers, SAT test practice programs, and access to the internet have become prerequisites to enrollment in a good college and getting a good job in the future. Those without a way to get a foot up into the system will be forever kept out. One aspect of the digital divide is that computers themselves must be made to appear inconceivably complex and incomphrehensible to the uninitiated.

The reality of the world we live in is that if 100 people represented the population of the world two of them would own personal computers. Owning a personal computer is much like owning an automobile and gives you bragging rights about the model and style that represents you. To the vast majority of the people in the world just owning one puts you in an elite group of rich and affluent people whether it is a clunker or the top of the line luxury model. Marketing of computer hardware and software is pervasive throughout the culture. Everyone would like the most beautiful, expensive, fastest and highest quality model in an ideal world.

Part of modern culture seems to be that people like to pretend, perhaps even to themselves, that they are so rich and important that money is of no object to them. If they can say that quality for value is not important to them because they only want the top quality most expensive option they get higher social status. Many people are therefore very arrogant about how wasteful they are with their computer. They are very proud of it and will tell you how they got the latest upgrades that they really didn't need but since it is all really cheap these days anyway etc. They will say that they tried the latest most wasteful new software and had to go out and buy a faster computer but didn't care because they are cheap. 98% of the people in the world think computers are too expensive to buy and most of the people who buy them don't really believe they are so cheap that no one cares about that. If you are talking about most of the limited resources in our culture it is not fashionable to brag about conspicuous consumption, but computers seem to thought of as an unlimited resource because of marketing.

I was first exposed to the difference between programming and computer marketing almost thirty five years ago. If you have been a programmer perhaps you have been there too. Your manager tells you, "This program you have written is too efficient. You don't understand the big picture. The client is spending $100,000 a year now to do this manually. Based on the runtime of the program in this form this program would accumulate only $5,000 in fees for an entire year. The client is now spending $100,000 and will be happy to have it done for $50,000. Go back and rewrite this program so that it uses ten times as much computer time so that we can charge this client ten times as much. This is a business and the bottom line is making money not writing efficient programs."

The nature of business management in the US is such that managers work their way up through the corporate structure by showing that they can manage larger and larger budgets. If you show an aspiring manager a way to reduce their budget they will know that they will be expected to live with that reduced budget again next year by a maybe not so understanding level of management above them. They also know that the one of their peers with the largest budget will most likely be the one promoted up to the next level where the budgets to be managed are even bigger. These pressures lead middle managers to pad their budgets and this is one of the driving factors in the computer industry.

IBM exploited this vacuum for years with managers being promoted for pouring money into mainframe accounts with constant upgrades of machines and operating systems to address a constant list of bugs. When I worked for Bank of America in San Francisco one of my managers maintained more mainframe accounts for employees who had quit than he had for employees who were still working for him. This allowed him to inflate his budget by about 20 phantom employees times $1000 a month to IBM for their mainframe computer accounts. I had three accounts and they were all still active three years after I had left the bank. My manager had given IBM $108,000 for my computer use after I was no longer an employee of the bank. Multiply that times twenty employees each for four thousand managers and you get the picture at the bank at the time.

As time moved on it became easier to get promotions in large companies for wasting money on Personal Computers. When I was a consultant to Pacific Bell I saw countless examples of managers spending hundreds of thousands of customers dollars on inflated budgets for computing systems to work their way up the corporate budget ladder. Managers were looking for packages that came in the largest boxes, with the most diskettes and with the largest pricetags then they would buy hundreds or thousands of copies that were not needed for any conceivable reason. One example were the 3270 terminals. They had many employees who used IBM 3270 terminals to talk to their mainframes. They replaced them with PC running 3270 emulation programs. This allowed them to continue to spend thousands of dollars per machine every year for needless hardware and software upgrades even though all of these users only ever ran one piece of software, 3270 emulation.

Whether you are talking about corporate America being marketed products that are intentionally puffed up for marketing purposes or individuals being marketed new computer hardware and software products based on style, status, and hype it is hard to deny that what sells are big boxes and big programs with lists of features that far exceed anything related to productive work. There is considerable concern both in the industry and by consumers about the diminishing returns for our continued investment as users in this kind of software.

The marketers tell us that if cars were like computers the cars we would be buying today would be absurdly cheap and absurdly fast, but it just isn't so. I got my first personal computer for about $1000 in 1975. That is still what they cost. The graphics are better. The computer is bigger and faster, and doing more complex things, but that is what you would expect after 25 years of progress. My first machine ran at about one hundred thousand instructions per second and my current machine runs at one hundred million, 1000x faster. My first machine was so slow that it would take several seconds to boot up and would sometimes just go away while executing system code on badly written programs and I could not type for up to twenty seconds. My current PC is 1000 times faster, but the programs seem to be 1000 times bigger and slower because it now takes about a minute to boot up and it still goes away for about twenty seconds sometimes while the OS and GUI do who knows what sometimes and appears dead and will not accept a keystroke for that period of time.

In this world of quickly expanding computer hardware and quickly expand computer software there seem to be very few people concerned with making computers efficient or getting the most value out of the computers we have, or the most productivity out of the programmers. It is more fashionable to claim that everyone (who is important) is rich and doesn't care about things like efficiency. If a program is inefficient they can always just go out and buy a more expensive computer to make up for any amount of waste.

In this world there are few people working on making computers simple to understand, simple to build, and simple to program. There are few people making programs that are easy to understand, easy to maintain, efficient and beautiful. One of those people is Charles Moore the inventor of the computer language Forth. Chuck Moore describes himself as a professional who gets personal satisfaction out of seeing a job done well. He enjoys designing computers and writing very efficient software. He has been working for nearly thirty years to come up with better software and nearly twenty years to come up with better computer hardware. His latest work involves unusually small computers both in hardware and software.

His ideas are very synergistic as both his hardware and software are as much as 1000 times smaller than conventional hardware and software designs. However many of his ideas about software design and programming style are not limited to his tiny machines. While it is difficult to map bloated software techniques to tiny machines it is easy to map his tight tiny software techniques to huge machines. There will always be problems bigger than our machines and there will always be people who want to get the most out of their hardware, their software, and their own productivity.

Chuck's approach to VLSI CAD is a good example of the application of his style of programming to a conventional computer. The approach and the design of the code used the tiny approach to get the most productivity from the programmer and the highest performance from the software on a conventional Intel based PC. Instead of purchasing packages of tens of megabytes of other people's code for hundreds of thousands of dollars Chuck wrote his own code in a matter of months to make it faster, more powerful and more bug free. He does the job more quickly with thousands of times less code. The size and performance of the program are quite remarkable and the methodology behind its design and construction involve more than a specification of the features of his language. It involves understanding how that language was intended by used by its inventor.

Chuck has moved most of this Forth language into the hardware on his computers leaving so little for his software to do that it is very difficult for people to see how his software could possibly be so small. He has refined his approach to his language until is is difficult for people who have been extending it for twenty years to see all he has done with so little code.

There are aspects of his early experiments with CAD that have led to great confusion about his software style. It has focused many people's attention on the number of keys on his keyboard or the size or number of characters in his fonts or the hue of the colors that he selects in CAD, the names he used for opcode and a long list of other distractions.

Introduction

Having spent the last ten years working with **Chuck Moore** on his custom VLSI Forth chip development I have greatly changed my ideas about Forth. I have moved on from the concepts that I first learned about Forth twenty some years ago and studied what Chuck has done with Forth in the last fifteen years. I looked over his shoulder a lot and asked him a lot of questions.

When the Forth community first began work on the ANS Forth standard the effort involved defining a Forth specification that provided common ground for different Forth users. The ANS Forth standard as I think Chuck would say was designed to cover almost all the variations on what Chuck had invented that everyone else was doing twenty years ago. There was never anything like it before, a sort of meta-Forth definition. But Chuck said he worried that this formalizing of a definition of Forth would result in a sort of crystallization of Forth. My concern was a different consequence of ANS which was that a new style of Forth programming seems to have evolved. Traditional Forth was on a real machine where there was a hierarchy from primitive hardware through abstracted code. There was always a sense of what were the simple fast primitive words were the best to get the most efficient code where that was needed. In ANS there is no such sense that the 10,000th word in the system is necessary any more high level or complex than the first since that is implementation dependent. Even though such a hierarchy of complexity will normally exist in Forth common practice in ANS Forth is to ignore this reality.

Chuck's advice regarding programming is often highly contextual. He will say people should not use most standard OS services rather you should write the code yourself. He says this because if you build your code on inefficient code you will have an efficient application and you will have to do more work to get it to work. At the same time the primitive words in Forth are also a set of standard services. On a real system you know the real tradeoffs regarding each of these services and can make informed decisions regarding which words to use. On an abstracted model of Forth (ANS) you cannot make these kinds of informed decisions. As a result ANS Forth programmers do with Forth what Chuck would advise them to do with OS services, they try to rewrite them themselves. Instead of using perfectly beautiful Forth words the way Chuck had intended them to be used 30 some years ago they rewrite their own version. In this case Chuck would not advise them to rewrite it themselves. I would often ask ANS programmers, "Why did you rewrite this word with all these pages of high level code when almost exactly the same thing is available in highly optimized CODE and is 1000x faster?" "Because that is the definition in the library in the system I normally use." was the answer.

Chuck and I were both convinced that this sort of abstracted approach to Forth might result in a new style of using Forth that would in turn lead to the ultimate demise of the language. We focused our efforts on building the fastest, simplest and cheapest hardware and fastest, simplest and cleanest software as an alternate future for Forth. Chuck's ideas about Forth have evolved through four stages in this time and I have generally been a stage behind.

After Chuck left Forth Inc. and began working on Forth in silicon he had the chance to start his approach to Forth again with a clean slate. He was happy with many improvements but did not stop experimenting after he did his **cmForth**. He moved on through the OK phase, the Machine Forth phase, to his current **Color Forth** experiment.

This document is not intended to be a programming tutorial. It is not going to present a step by step explanation of how one programs in the style that Chuck Moore is using but will present an overview of what he is doing and why. There is an older document describing **Forth and UltraTechnology**

Table of Contents

Preface

Introduction

Table of Contents

Chapter 1

- Context
- Object Code

Source Code
The First 10x
Extending Forth
Dispelling the User Illusion
Chapter 2

cmForth innovations
Virtual Machine Improvements
Portable Simple Native Code Compiler
Stacks, Words, Blocks
More Interpretation
Delayed Interpretation
ICE
Code Metrics
Avoid Abominations
Remove at Design Time
Chapter 3

Rethink the Problem
What to Throw Out
The Problem of Recursion
Focus
Attention
Example in Practice
Translation of Code Between Languages
Scaling
Wordlists
Addressing Constructs
Overflow and Underflow Errors
When to Inline Rather than Factor
Decision Trees
Conclusion

Chapter 1

Most of the Forth community have had little exposure to the evolution of Chuck's Forth for the last fifteen years and have now become deeply entrenched in their habits from twenty years ago. Chuck has lamented that no-one has published a book teaching people how to do Forth well. Chuck has seen how other people use Forth and is generally not impressed. On this page I will discuss aspects of the Forth language as I currently see them and lightly cover the subject of good Forth programming.

A Definition for Good in the Context of Forth Programming

What is good Forth? What makes one Forth program better than another? Well of course it depends on context. The first thing in that context to me is the computer. Real programs run on real computers. By that I mean real programs are implementations not specifications. You can specify the design of a program in a more or less portable form or you can specify the details of an actual implementation of that program more explicitly. In either case I am talking about two aspects of the program, the source and the object. I will discuss what I mean by good source and good object code.

Good object code is pretty straightforward. It is efficient in terms of system resources, it does not consume resources excessively. The particular resources for a given system and a given program will constitute a different balance of things like memory use, speed (time use), register use, cache use, I/O device use etc. On many architectures there is the tradeoff between code size and speed. Up to the point that cache overflows longer sequences of unfactored instructions will execute faster so many compilers perform inlining of instructions. At the point that cache overflows things can slow down by an order of magnitude and if the program expands to virtual memory paging from disk things will slow down by orders of magnitude.

A little smaller, a little bigger, no big deal. A little faster, a little slower, no big deal. But when the ratios become quite large you really need to pay attention to the use of resources. Since there are so many layers that all multiply by one another in terms of efficiency if a system has ten layers that each introduce a little more fat the final code may see a small fraction of the total CPU power available. Programmers need to remember on most modern machines the CPU is much faster than Cache memory, cache memory is much faster than onpage DRAM access and offpage DRAM access is much slower than onpage. Regardless of other factors the way the program organizes data in memory and how it is accessed can easily effect program speed by more than an order of magnitude. What is marketed as a 100Mhz PC can easily be slowed to 10Mhz by slow memory access depending on the program. It can be effectively reduced to almost nothing when the software goes away for 20 seconds at a time unpredictable to do some system garbage collection or something. From the user's point of view for those 20 seconds the machine has 0 user mips. Programs slow significantly when the program or dataset is so large and access to it is so random that the worst case memory time happens a lot. This and much worse is what happens as programs grow and spill out of cache and out of available memory. To avoid this keep things small.

In some cases, such as scripting languages, fat is not an issue in terms of code efficiency. It remains an issue in programmer efficiency however if that fat is a source of bugs just like lean code only moreso. Excessively fat programs can easily be excessively buggy and unstable because the bugs will be hard to find in all that fat. Also if a program is grossly inefficient at runtime it may not be as important as the time spent writing it. There are many one-of type of applications where big and slow is not an issue such as trivial scripts that only run once in a while. But for system software it is very important that object code not be too inefficient because other things are built on top of it.

Of course some people would say, who cares, just buy a more expensive and faster computer to make up the difference. Sometimes that makes sense. But for those who have been in those BIOSes and system software and seen how bad it can get it seems like a shame to see people being forced to waste 90% of their investment in hardware or software because it means someone gets to charge more money. In this sense the inefficiency fuels the planned obsolescence and forces people down the expensive upgrade path. It's good for you if you own Intel or Microsoft but otherwise it is a concern that has spawned the growth of PD software like Linux.

Good source code is a bit more difficult to define. It should be clear, easy to read and write, easy to debug. Again a little smaller, a little bigger no big deal. But computer languages are more different than one another than human languages. When people see a language that is considerably more brief or verbose than the computer language that they are used to their immediate reaction is usually I can't read that, it's too little or it's too much. To compound this variation in point of view the visual layout of the source is a big issue. The attention of reader is directed by code layout and this is also a big factor on how readable the code will be. If the comments are in a language that you don't read they don't help. If they are in a font that is too small to see they don't help. If they are printed in a color that you can't see they don't help. Fortunately some vision problems are correctable but these are issues.

For some people the code layout must be pretty. This may be more imprint to some people than code contents. I can't relate to that myself. To me the layout is simply there to direct the attention of the reader. You are not trying to give them an esthetically pleasing experience so that they sigh when they look at the page and don't bother to read the contents. If you follow code layout rules they are there just to make the code clearer.

Chuck has switched to color in his latest Forth as a replacement for some of syntax and words that he had not already eliminated. **: ; [ ] LITERAL DECIMAL HEX \ ( )** are some of the words that Chuck has replaced with color change tokens. What I find most interesting about this is that when reading the code a different part of your brain is engaged in seeing the organization of the code into words and what the compiler and interpreter are going to do with the code than the the part of your brain that decodes the meaning of the words. It seems to free the part of the brain reading words to focus on the words more clearly because there are less distractions. Mostly Chuck has replaced some layout information and some Forth words with Color. Besides making the Forth, small and fast, as Chuck puts it, it also makes it colorful. My own experience with his Color Forth is that the result is easier to read code than conventional Forth. But until I have tried using it myself I am not ready to make a final judgment about that.

As I have said, prettiness is more important to some people and beauty is in the eye of the beholder. Some peel think a system described on a couple of pages clearly is beautiful in itself just as a concise equation in Physics. To another a listing that looks like a telephone directory is beautiful. People will never agree about what looks best. Chuck has limited detail resolution in his vision and complains that he can't see small fonts on the screen. He uses large fonts so he can see them and as a consequence he only has short lines and small definitions. Other people have screens with 256 characters on a line and some very long Forth definitions. Chuck complains that he can't see those small characters and that the code should be factored into smaller pieces. (when the code is printed in a larger font Chuck has also complained that he still couldn't read it because often it would begin with lots of words that had been loaded from a user's libraries that are essential for the author to write anything but which can only be described as extensions to Forth. If you know all of these persons extensions you might be able to read the code.) This same author complains that he is color blind so Color Forth doesn't work for him, even if he were not color blind the lack of layout and spelling rules would make it unreadable to him. Of course color has been substituted for layout and some words in Color Forth. Chuck feels color is good substitute for layout and some words, other people don't or haven't tried it.

As I say the layout issue is very personal, one person may have a couple of rules for layout and someone else may have about as many rules for spelling and code layout as another person needs to define the Forth system. My stance is that this is a matter of taste and I have my personal style and I can read either extreme of code. The code with pages of layout and spelling rules looks nice and if you cross reference all the words that came from the user's private libraries the meaning is clear. I find Chuck's Color Forth very easy to read too. I think it is easier for me to read but part of that is the same reason that a 25K source is easier to read than a 25M source.

Size becomes a significant factor when it comes to being clear, easy to read, easy to write and maintain etc. when the numbers ratios become quite large. Very small programs can be read quickly but may include subtleties that elude easy perception on the surface. They may need to be read more than once, or they may require more documentation than the code itself to be clear. If code is too dense it will appear as nothing except meaningless cryptic symbols unless it is studied in great detail. If code is too verbose it may appear as perfectly clear line by line but impossible to view because of size. Yes, I can read source code, but no I can't read 25 megabytes of source and keep a picture of it all clearly in my mind.

So my the definition I am use here for good source is something that conveys meaning to the programmer effectively. I would say text, but it could include graphics in visual programming or Color, Font styles etc. Just call it source to distinguish it from a formerly sourceless programming environment like OK.

The First 10x in Forth

Forth had a surge of popularity in the seventies when FIG was distributing source and alternatives were limited. Many users who discovered Forth at that time reported elation at the increase in their productivity. They wrote programs faster, they debugged them faster, they maintained the more easily. They reported that they could write much smaller and much faster programs that could do much more than the ones they could write before. But when they reported that they had seen a 10x improvement after switching from ... they were often dismissed by mainstream programmers as kooks because that just seemed too good to be true to many people.

Those who were there know that the 10x is not all that remarkable and is really due a bunch of numbers that when all multiplied together equals 10. No single thing gave these programmers a way to be ten times more productive, instead it is all the factors multiply by one another.

The reasons have to do with the design of Forth. Stacks, words, blocks. Having the data stack for data is simple and beautiful way to handle and pass data within a program. It introduced less bugs than environments where programmers were working with lots of named variables or where they had to juggle register use by hand in assembler. The separation of data and return stacks made factoring more attractive. If you don't have to construct stack frames and move data in and out of function call's local variable data space before you call something you have less overhead in calling a function and can factor the code more extensively. **"Factor, factor, factor. Factor definitions until most definitions are one or two lines."** is Chuck's advice.

Factoring was a key to debugging and maintaining the code. Well factored code is easy to debug and maintain so the programmer is more effective with their time. Factoring also helps achieve the desired balance between memory use and speed for a given machine since memory and processing power are always finite.

Programs were often performance limited by their interaction with mass storage as they are today. Forth provided the BLOCK mechanism as a very simple form of virtual memory and close to the metal mass storage access. By using BLOCKS where they could for data instead of more complex file access programmers reported speeding up parts of their programs by 100x as well as making them smaller and simpler.

Programmers often also reported that with Forth they could change anything. They didn't spend large amounts of time caught in bugs they discovered in someone else's compiler, or in elaborate work around schemes fighting with their software as they had before. If they wanted to change something they just did and moved on to other productive work. They didn't get stuck like they often had before.

So armed with software that was smaller and simpler and easier than what they had before and with an interactive modular develop and debug methodology that was more effective in the integrated development environment they were happy. They were delighted to have seen improvements in their productivity as programmers, their understanding and their freedom to do what they wanted to do. They also turned off a lot of people who didn't want to believe that these people could actually have all this stuff and made comments about how this was too good to be true so Forth must just be a religion or cult or something.

So far everyone has said, yes, yes, we all know this ancient history of Forth. So far everyone has been with me and mostly agreeing. So let's get to the more controversial stuff.

I begin with this history because it is my opinion that this is as far as most people got before they headed back toward more conventional programming practices for various reasons. Little by little as people added their favorite features from their favorite languages to their Forth and fought to standardize the practice Forth became bigger and more complex. Bigger computers and growing libraries of code made it possible to easily compile bigger and bigger Forths.

When Forths were small and simple they didn't take much source code. There weren't too many words. Even on the slow systems of the old days a simple linked list of the name dictionary was sufficient to search things quickly. As systems became larger and larger dictionaries became more complex with wordlist trees and more complex methods of searching the more complex dictionaries were introduced which introduced more complexity. In an environment of spiraling complexity in the popular operating systems and GUI Forths expanded their interface to keep up. Now the glue between Forth and the user interface could be hundreds of times bigger and more complex than a complete Forth system in the old days.

Some of these Forth systems are advertised as having the best compilers that produce the fastest code. What they don't tell you is that it may be true given that you are ready to accept a 90% or 99% slowdown to move into that environment in the first place. If you choose to mount your Forth under a GUI that takes 90% of the CPU power and leaves 10% for your Forth you may need that optimizing compiler even on a fast computer. We have ported the chip simulators to various compilers. We moved from a 16 bit DOS environment to a 32 bit Windows environment to get higher performance. When we hit the OS wall we still wanted more speedup so we ported back to the 16 bit DOS environment where we could get out from under the API load. We were able to speed up the program 1000x times by switching to a Forth that wasn't crippled by its Windows interface. What is interesting is that the program runs 1000x faster in a Windows environment by ditching the Windows Forth. We have a strong incentive to replace the many megabytes of OS code with a couple of K of reasonable code to get the same functionality. We prefer programs that are 1000x smaller and 1000x faster and easier to write and maintain etc. If you are stuck in an excessively complex environment using Forth gets you out from under some of the complexity facing other people, but only a tiny bit of it.

Complexity demands more complexity. When the source code gets really big and complex it begins to demand things like version control utilities. Now between the huge files and time spent on version control jobs become too big for one person so we split them up and assign a team. Now we need a team of four. Now we need a more complex version control system with multiple user access. Now programmers are spending more time with the complexities of version control and other people's bugs that four isn't enough so we expand the team. Diminishing returns is the obvious result.

Many commercial and PD ANS Forth implementations Forth have become as complex as 'C', or extensions to 'C'. The ANS standard went beyond the Forth core into extension libraries and it was common practice to start with everything from the last twenty years. We had Forths that were hundreds (or thousands) of times bigger and more complex than early Forths. They still supported the factoring, and the interactive nature of Forth development so they still had some the factors that made up that old 10x that we loved in the old days. But often now they were carrying megabytes of baggage and users were dealing with programs as large and complex as many other languages. Forth had changed so much that many systems require a full page or more of code to build a hello-world program thanks to the use of dreadful APIs.

There were traditional Forth programmers and new Forth programmers who could use these environments in a similar way to what they had once done but the common practice was to introduce coding style, layout, and libraries plucked right out of other languages. Common practice became very unForthlike and in particular beginners were often exposed to such examples in places like c.l.f. between the debates about who could write the wierdest code to break the outer fringes of the ANS Forth standard.

Chuck's view of programming, as I understand his description of it, is that there is a problem, a programmer and his abstraction and the computer. Forth was there to let the picture be as simple as possible and let the programmer map the solution to the problem to the computer.

Problem

Programmer with abstraction of problem

Computer

and this leading to a solution that looks like this.

User

Programmer's simple implementation by abstraction of the problem to the computer
---
Computer

This was Chuck's original idea of Forth even though in the old days the normal picture was not as complex and layered as it has become today. There were only a few layers between the programmer and the computer in those days but that was the problem that Forth was suppose to avoid. As the layers have become more numerous and deeper it has become even more important to let Forth avoid that problem.

As each of the layers of abstraction were added to the model and common practice over the years we were told that each would result in smaller, simpler programs because they would not need their own copy of things we standardized on. Programmers were suppose to become more productive and systems were suppose to become easier to understand and software would be easier to write, there would be more code reuse etc. Like cooking a frog in a pot the water the pot got hotter and hotter without people noticing who was coming to dinner until most of the problems most people face were introduced this way. People complain now that they spend more time looking for some code to reuse this way than they used to spend writing code when they used to do that. Chuck on the other hand has learned how to be more productive and write better code faster.

Chuck wants there to be nothing in his way. Chuck wants to make the computer simple and easily comprehended so that no extra layers of abstraction are needed to get at it or the problem. Chuck wants to make the solution simple so that it easy to write and efficient and has no extra layers of unneeded fat. Chuck seeks a simple efficient abstraction of the actual problem to the actual computer.

Chuck does not like the idea of a generalized OS providing a thick layer of abstraction that can introduce unneeded code, unneeded complexity, and inefficiency. He will support the idea of the abstraction of an OS but not one for everything. He and I would agree that in many environments there are layer upon layer of abstraction, that introduce complexity.

The people coming into computing in these times are being taught that the picture below is reality of a computer. They face enormous problems as a result. Almost no one gets to deal with the simple reality of the problem or the computer but must deal with the complexity of a thousand other people's abstractions at all times.

Problem

Programmers's abstractions of problem(s)

Programmers's abstractions in software (example: OO w/ late binding)

Programmers's abstractions of software reuse (general source libraries)

Programmers's abstractions of optimizing compilers knowing more than they
---
Programmers's abstractions of the computer GUI API

Programmers's abstractions of the computer OS Services

Programmers's abstractions of the computer BIOS

Programmers's abstractions of the computer architecture ('C')

Computer (too complex for all but a few humans to grasp)

These are two very different points of view. Chuck has said that he would like to Dispel the User Illusion. He means that the user has the illusion that all these layers of abstraction **ARE** the computer. If they could see beyond the illusion to see only the simple problem and were only faced with mapping it to a simple computer things stay simple and simple methods work. The majority of problems are avoided this way.

Those who have been working on making Forth more mainstream, extending it, and merging it with 'C' libraries and popular APIs have applied Forth in a very different way than Chuck. What made Forth popular twenty years ago was that Forth provided a simpler model and made programmers more productive because they weren't trapped behind so many barriers introduced by other environments. They could do things the way that made the most sense not the way they had to be done.

Chuck originally created Forth to avoid problems introduced by unneeded abstractions. There was the abstraction of a Forth virtual machine and the expression of a solution in terms of that abstraction. Chuck has spent years simplifying and improving the virtual machine and has moved that abstraction into hardware to simplify both hardware and software design. He has a simpler virtual machine model, implemented in hardware on his machines, and a simple Forth environment implemented on top of it.

In many discussions that I read in c.l.f someone will ask how other people would accomplish such and such. My first thought is usually something about how it couldn't be much simpler than what we do. **Acolor fontcolor !** to change the color of output in a numeric picture. What does it take? A store to memory, a few nanoseconds is the answer when you keep things clean. Other people will post tens of pages of detailed code that they need because of the bizarre behavior of particular layers of their layer upon layer of abstraction introduced problem laden environments.

People have said that without all this abstraction the general purpose OS could not run on innumerable combinations of cobbled together systems made of boards and cards and drivers from a thousand different vendors. This may be true although only in isolated cases will it sort of run anyway. It is also true that computers don't have to be built that way. They can be built with logical, simple, inexpensive but high performance designs. The problem is that there is a cost to carrying around drivers for thousands of computers when in reality you are always only using one set. Neither hardware nor software have to be built that way. The problem is that the number of bugs and related problems or the amount of waste of resources can cripple the computer and/or the programmer.

With so many people using Forth today as a sort of scripting environment on top of every generalized service and abstraction as everyone else the common practice in Forth was no longer 10x compared to other ways of solving those same problems. Meanwhile I have been watching Chuck very closely. He seemed to still have a 10x up each sleeve that I saw very few other people using. He had a very different style of using Forth by continuing to explore in the direction he had been headed with Forth originally while most of the Forth community was going in the opposite direction. What are these other 10x factors?

Thoughtful Programming in Forth Chapter 2

Thoughtful Programming in Forth Chapter 3