Thoughtful Programming
Chapter 3
Third 10x

Now that brings us to the third 10x that Chuck has up his sleeve. This is the one that I found most elusive. This is the stuff that he has not explicitly and repeatedly presented to the Forth community. It is the stuff that is not obvious to everyone and which you have to dig in to find.

I can only assume that given the reaction that Chuck has received for the advice he has given on good programming style that there is no point in his trying to go on to the advanced stuff. I know I have felt very much that way over the years. When I have given presentations I always want to get into the interesting stuff like how to implement a GUI in a couple of K or how to compile English language rules into optimally efficient executable code structures or how to parallelize Forth programs easily but we rarely get beyond reviewing the most basic and simple facts to get started. People seem to begin by trying to map the ideas about the chip and Forth to other languages and environments and architectures. As soon as we try to get going people say, "that's not possible" because they are thinking 'C' or thinking RISC. We spend most of our time simply explaining that the starting point and the Forth perspective and that our programs and our hardware match very well because the hardware is 1000x smaller and cheaper and lower power consumption and we only need about 1/1000 the code so it fits. It is simple and easy and productivity goes way up when you are not dealing with the 99.9% fat and all the unneeded complexity.

The explanations often go like this: "I don't think it will work, you can't handle the ... problem." "We don't encounter the ... problem in this approach we avoid it." "But you can't handle the ... problem." "We don't encounter the ... problem when going in this direction." "How can you handle the ... problem?" "That is what we are optimized and designed from the ground up to do." They have a very hard time seeing that we face a different set of problems by simply avoiding so many unsolvable problems.

If I post a fact in c.l.f such as the time for a specific processor to run a specific program so that people can compare it to other processors it gets lost in the noise of all the people who post what they think are corrected estimates based on their perspective. I have asked people why they would say in c.l.f that I was not being honest in simply reporting results and ask them where they get the information that they published. "Well it was just my estimate of how well F21 would run SpecFP in Unix." If one has read what Chuck and I have said over the years they would know that either the person doesn't have a clue or is trying to be an intentional deceptive as possible.

As I say the last 10x is the most elusive and the stuff that I don't think Chuck has explained repeatedly although it has been put in front of us. Of course those who deny that it is there are never going to find it even when it is pointed out to them. I imagine that when Forth Inc. trains people that they provide them with some of this stuff. I have seen articles in FD about some of the techniques but it tends to get lost in all the stuff about making Forth look more like other programming languages.

People try to distort and trivialize Chuck's by saying that he has removed everything that is needed from Forth and they don't understand why. But Chuck didn't just remove things to remove things. He removed things when he found a simpler way to solve the same problem that something was there

solve. Often to solve a problem you have to add x-y-z. But then x-y-z introduces some more complexity and as a result you have to then also add a-b-c and d-e-f. Now everyone gets used to x-y-z and never considers any other way to solve the problem that x-y-z has solved for everyone for so many years. They also never consider that a-b-c and d-e-f may also be viewed as problems that are begging to be fixed by being removed. Otherwise as time goes by they become bigger and bigger problems and as code is developed and added to the system they will lead to more complexity.

So Chuck will rethink the problem. He will find a different way to do x-y-z, or better yet to avoid the problem that x-y-z solved. Now the x-y-z solution that everyone has been using for some problem is no longer needed because something better is now available. It gets done because now he can also remove a-b-c and d-e-f since they were just needed to support x-y-z. Things become simpler, clearer, easier to use, easier to maintain, because unneeded fat has been removed and there are fewer complications.

But people who have been doing x-y-z for twenty years are horrified at the prospect of not using it. They cannot imagine Forth without it. None of their code would run without it. They would have to rewrite code to not use x-y-z and writing code is hard for them. They also cannot imagine living without a-b-c and d-e-f because they once again their old code is full of that and they never considered doing it any other way.

Chuck doesn't just throw stuff out because he is compelled to minimize. He is compelled to experiment and improve his code. He is not against throwing something out if he finds a way to be more productive without it. He throws stuff because he can more productive by doing things in a better way, cleaner, clearer, simpler. He does not think that he came up with the perfect language 30 years ago any more than he feels he has the perfect language now. He will always want to change and improve what he is doing and he will always be looking for new ideas to make it better. He has been on a clear path to make his language, Forth, smaller, simpler, easier, more powerful, faster, less buggy, more maneuverable, etc. But each time he finds a way to improve something, through extensive experimentation and pragmatic analysis many people have knee jerk reaction and say, "He took out what? I can't live without that! What is he thinking?" But rather than ask what he is thinking and ask him to explain why it is better to do Forth without x-y-z many people just say, "x-y-z is standard practice by the average programmer and is part of the ANS standard. I don't want to even consider the idea of removing x-y-z and I have interest in listening to the arguments. I will just post stuff for other people about how I know that Chuck knows that x-y-z is really good and that he is just trying to mislead people.

Look at Chuck's **cmForth** from a decade ago if you haven't. Why was SMUDGING removed? That's non-standard! Why were immediate words removed? The answer is pretty obvious, he found a better way to solve the same problems those things were there to solve, they were no longer needed and so were just in the way. He as explained the reasoning but only a few of dozen people have followed his logic in the design of cmForth. Fewer followed his transition to Machine Forth, it sort of slipped through the cracks. Very few had any interest in OK, it wasn't Forth. OK as originally called 3/4 (three Forth) because Chuck considered it subset of Forth at first but later decided that it wasn't Forth. By the time he had moved to his **Color Forth** Most Forth enthusiasts had lost any interest in what Chuck was doing. He wasn't doing ANS Forth like everyone else and some of the ANS Forth people even felt that Chuck should not even use the term Forth any more because they were now the official owners of

the term and Chuck wasn't doing what they said was Forth.

The consequence of removing SMUDGING was that compiling was simpler and recursion became automatic. As a result if Chuck wanted to redefine a word he would have to do something like

```
' MYWORD
: MYWORD ( redef) ... COMPILE, ...
```

because **MYWORD** would be recursive call to the new **MYWORD** without SMUDGING of the name until the definition is finished. But Chuck said he would just redefine it with a new name anyway because that is simpler.

Chuck also added tail recursion a decade ago to his Forth. This only gave him a small speedup on many words by converting the last call into a jump rather than compiling an **EXIT** in the **;** A little speedup here and a little speedup there and if you keep at it you see overall speedup. In the same way a little fat and sloppiness here and a little fat there and soon you see a lot of fat accumulating.

By combining this tail recursion with the auto-recursion provided by removing **SMUDGE** from the system this provided a looping construct. With this new simpler control flow mechanism Chuck didn't need all those others. He had already tried replacing **DO** with **FOR** a decade ago and now able to remove other looping constructions now that he had a simpler more efficient mechanism available.

When Chuck removes stuff from his Forth that other people use regularly in their Forths they wonder how he can live without ... or ... and they don't seem realize that he didn't just take stuff out randomly or for no reason. He took stuff out after many years of extensive experimentation and consideration because either it was replaced by something that he considered better or he felt that it was just getting in the way and no longer needed at all.

People assume that since Chuck has refined his Forth down to about a 1K object that this means he has just stripped his Forth down to a 1K kernel that will boot like in the old days and that he is going to compile a complete Forth system on top of the 1K before he starts an application. This is wrong. The complete Forth system is 1K, and the reason for that is maximize Chuck's productivity. What stops people from doing what they need to do to solve a problem is all the time spend solving all the related sub-problems that pop up as a result of complex interconnections between components. To maximize his productivity Chuck minimizes the number of these side problems that pop up. Keep it simple, and don't get to where you are spending 90% or 99% or you time dealing with related sub-problems. Avoid unsolvable problems, don't waste your time trying to solve them.

The approach that Chuck has taken is to focus on the task of solving a problem at hand. This means facing the situation of a real program. Unless you are writing a book about programming theory if you are dealing with a program it is the implementation of a program on a real machine. There are many real issues that come up when one faces the situation at hand.

You have a problem, and are facing the implementation of a program in the real world. The approach Chuck takes is to think through the problem well before you design the app. Part of that design involves issues around the real platform on which the code will run. Face the problem, think about it until you can picture the solution as about 1K of code or less. Until then you don't really understand it.

If you assume that the program is going to be megabytes of code it is very intimidating. The first thing people look for is ways to crank out code and do as little coding as possible. They can't possibly write a megabyte of real code themselves so they look for code in libraries. They paste in the code, and as much as possible and then go from there. They see no alternative. There is just too much code to deal with to study it in detail.

Sometimes the prospect that programmers are so unproductive and programs are so big will make development costs look more attractive if spread accross multiple platforms. The idea is good and sometimes it is better to only spend the minimum on coding and live with inefficient code that is mostly portable to multiple platforms. That is the case when the programmers are crippled. This is not the mindset that Chuck would advocate. Don't base all the plans on the restrictions imposed by crippled programmers.

Once you have made it through the second 10x you are no longer facing the unpleasant prospect of only being able to cobble together systems out of large collections of code written by lots of different people. You have the option of doing the same job with a small amount of well thought out code. The code is small so it easy to focus on making it efficient. The code is small so it is easy to focus on making it match the real details of the platform on which it runs. Writing megabytes of code requires that you paste in a lot of stuff without trying to understand too much detail, not very Forth-like. Writing kilo sized applications exposes all levels of what is going on in the code and provides the anything can be changed ability that accounted for the first 10x. The applications can be easily ported and easily maintained and easily improved.

If an application deserves to be written it deserves to be written well. It deserves to be well thought out and well implemented and perhaps even rewriten and reimplemented. Why? Because you will learn! You will learn how to make it better and better. If you paste stuff in in an effort to avoid thinking about the details do you really think your programming skills will improve? Do you really think you reached perfection on the last try or that you cannot learn by rethinking the problem?

One of the factors for that third 10x is thoughtful or mindful programming. Pay attention to what you are doing. Don't try to avoid thinking. Programming with thought really is more fun and more rewarding than programming without thought. Now I realize that this bucks the trend in modern software. Popular software keeps trying to dumb languages down further and further so that programmers will be a commodity resource. Any programmer given the same libraries and same tools will cobble together more or less the same mess. It is a good way to turn the programmers into replaceable parts. But who wants to be a replaceable part?

So you really think through the problem. You study examples in libraries. You try different experiments and models and compare features before you jump in and design code. Where do you jump in? I think you think through the problem from top to bottom and from bottom to top a couple of times. Then you look at the most important part the bottom. Most of the time in the profile will be spent at the bottom. The code at the top may execute so infrequently that it doesn't matter whether it is system style code or script style code. But the lowest level code, system level if Forth is the OS, or OS and interface code otherwise spreads its fat over everything else. If anything deserves attention this is where you start.

What routines execute most often? Where will the focus on efficiency be placed? This is the very bottom of the design. At this stage I advise programmers to study the data structures that are used by the program very carefully. There are some very important issues to the code design, data structures, ordering, and scaling. How do you design the data structures? You examine what they contain, how they will accessed, and how they will be manipulated. It is not that much different than planning the data flow on the stack. If it is well planned out things are often there whenever they are needed because of the planning at design time. When they are not in the right order you need stack or data structure gymnastics to manipulate the data. It is very important that the fat be removed at this level. The order in which data elements are accessed in the program should be designed so that as often as possible in the most frequent routines when a data element is accessed the pointer is left pointing at the next element with no overhead. No manipulation of the data pointers is needed when this is applied in as many steps as possible in the problem. Access to the data on which other routines depend can be sped up several times with something as simple as that. It is no different than carefully planning the stack use so that no stack gymnastics are needed.

I can think of an example where the common practice was applied by an ANS Forth programmer who chose to ignore all of the above advice and generate more portable code. Everything else was built on top of the data structures. The program was a translation of a program from a library from 'C'. The first step was to paste in 'C' like data structures. The second step was to make a copy of the data structures in the 'C' program and access them in exactly the same order as in the 'C' program. Worst of all the Forth implementation of 'C' structures used late binding so instead of pushing stuff from runtime to compiletime and from compile time to design time this was pushing stuff from compile time to runtime. The pasted code had not been examined closely. Even a novice Forth programmer would have cleaned it up and sped it up a couple of times if thought had been applied. So it gave up about 2x by not removing lots of visible fat, 2x for using the ordering in the 'C' program without question, 50x for using late binding and 50x for using ANS Forth rather than Machine Forth. This was far more than 10x when you combine the factors, more like 10,000x. I patched the code, ran comparative benchmarks and discussed the issue again in staff meeting. We had been assured that any code pasted in from FSL would get cleaned up appropriately for an embedded target.

Translating programs from one language to another can be done mechanically or by studying the implementation in one language and improving on it in the next. Don't translate when you have the opportunity to rewrite and improve. You can spend your time carefully designing code or trying to debug and improve thoughtless code. Translating without thinking is not thoughtful programming. Don't be afraid to think about the problem and what you are doing.

Another factor for getting that last 10x is scaling. Most Forth programmers don't take advantage of the tricks of scaled arithmetic. Their programs deal only with full integers or floating point. Scaling is important because it may allow you to take advantage of my favorite mathematical operation, cancellation.

Scaling is something that takes planning, not unlike the ordering and arrangement of important data structures. Precision and accumulated error are some of the considerations. **The real value is that when using scaled math you may have the opportunity to carefully scale the factors so that the most common operations are simpler mathematically or logically than they would be without careful scaling.** In other languages you may write an infix algebreic equation and let the compiler sort

out the operations to do it. You can add a routine to do that for you in Forth also.

Normally we put in the equation as a comment and sort out the sequence of operations to implement it outselves. In doing this we are used to refactoring an equation to simplify its calculation. In the same way that expressions can be simplified to remove terms some equations can be juggled to use operations that are simpler to implement than others. Shifts and adds can replace multplies and divides or more complex operations when the values are scaled to suit key constants and key calculations, or terms can be combined or made to drop out altogether. The program will execute this equation N millions of times, by scaling it this way these multiples become **1***, this divide becomes a **2/**, and these terms drop out. Of course we need to convert back to another range of numbers when we are done with all the calculations in the main loop so you rescale again when you are done. If you don't think this way you can't take advantage of this type of thoughtful programming technique.

Chuck showed me the equations he was using for transistor models in OKAD and compared them to the SPICE equations that required solving serveral differential equations. He also showed how he scaled the values to simplify the calculation. It is pretty obvious that he has sped up the inner loop a hundred times by simplifying the calculation. He adds that his calculation is not only faster but more accurate than the standard SPICE equation. In the article about OKAD from More on Forth Engines Volume 16 Chuck mentioned scaling of units for his transitor model. He said, "**I originally chose mV for internal units. But using 6400 mV = 4096 units replaces a divide with a shift and requires only 2 multiplies per transistor. This pragmatic model closely fits the measured IV curves. A (spreadsheet style) display exists for manually fitting parameters.**" Even the multiplies are optimized to only step through as many bits of precision as needed.

The third 10x comes from putting particular attention at design time to these issues of how to move things from runtime to compiletime and from compiletime to designtime. **The third 10x is to some extent a recursive application of part of the second 10x.** Just because you have a solution does not mean that you should not consider trying a better solution. This is most important in low level code.

Wordlists are a feature of Forth that links multiple lists of words to be searched in the name dictionary. One list of words for Forth, another for the assembler, another for the editor or at least that is the way it worked in the old days. ANS Forth provides a mechanism for constructing and managing wordlist trees in the name dictionary. I have noticed that this has often become a very abused feature of Forth. The multiple ordering of the wordlists to do different things leads to the problem of the same names doing different things in different wordlists and confusing the programmers as well as other problems. I have seen this carried to such an extreme that the programmer has constructed more wordlists than Chuck would write words! The worst case I have seen is the use of wordlist to implement classes in a object oriented Forth. As I say before the words in Forth are a primitive form of object if they don't do it for you you can use the **CREATE DOES>** construct for more object functionality. When systems overload the concept of OO classes onto a wordlists I think it qualifies a wordlist abuse. It introduces so much complexity.

Chuck has recently said that he has removed wordlists from Color Forth. With a 1K sized Forth that can compile applications in a click and forget them in a click he isn't dealing with code that requires being spread out over a wordlist name tree. Removing so much complexity from other places made Chuck feel that wordlists were just not needed to deal with the complexity that he used to deal with.

Removing wordlists is also one of the techniques that greatly simplified other words in the system and allowed Chuck to build a 1K sized Forth that can compile applications in a click.

There are several improvements that Chuck has added to his newer Forth virtual machine model. One of them is the address register and the other is the circular stacks. Chuck has explained that hardware considerations aside the idea of the address register was that the Forth words **@** and **!** (fetch and store) were clumsy at the top of the stack and were based on smaller atomic operations that the programmer could take advantage of. **@** was broken into two operations**A!** and **@A**. Likewise **!** was broken into **A!** and **!A**. One advantabe of this approach is that the contents of the addressing pointer can be preserved accross words independently of the data stack. Another advantage is the use of auto-increment addressing memory access opcodes. The use of them permits auto-incrementing items within loops of a program using less code and less time. There are some other consequences as he has said as to what kind of code falls out when you write for this sort of virtual machine. All the people that I worked with reported enjoying using the style and felt empowered with these techniques to improve their Forth code.

I advised the programmers who were using ANS Forth only to experiment a little with addressing this way in their high level code and gave them a simple set of definitions with a variable named A used for addressing. Code written in that style and with **BEGIN**s instead of **LOOP**s and other more high overhead words that do the same thing would result in a simpler and clearer style and make it trivial to port their ANS Forth code to Machine Forth on the project to get a large automatic speedup on this system

Another feature of Chuck's new Forth virtual machine model is his circular stacks. They greatly simplify the construction of his hardware and make the stacks in his architecture faster than general purpose registers in other architectures. They also greatly simply Forth. Chuck has said that stack overflow and underflow errors have always been a problem not just for Forth but for everyone. The problem is that they are destructive. Unanticipated errors happen but then the return stack gets corrupted errors can compound and systems can crash. Overflowing stacks can corrupt code and other data structures causing systems to crash. Hardware and software designers have put a lot of attention into managing these errors with elaborate hardware and software. That elaborate hardware and software adds complexity in other places like compilers and applications and that leads to more bugs. Chuck wanted to find a simple solution that wouldn't introduce complexity. Having the bottom N elements of the stack as a circular data structure meant that there was no arbitrary starting point. When you were done it was empty. You never had to empty it out before using it again, it was always at the start if you wanted it to be. If you program had bugs the worst thing that could happen on the stack is that stack data would be corrupted. That kind if error is a lot easier to deal with than corrupted code or memory structures. It also means that a Forth system does not have to deal with complex hardware mechanisms or complex software mechanisms the problem is either avoided or minimized as well as any approach can make it almost for free.

Now Chuck says to factor, factor, factor. But in fact he also sometimes does just the opposite, he inlines his code! But not all the time, only were unrolled inner loops will improve the performance of a critical routine. It is a common practice in other languages to allow the compiler to do this for you. Chuck however simply specifies it explicitly when it is better than factored code for the problem at hand. He doesn't do it with complete applications, only a few routines.

These inlined code unrolled inner loops also give Chuck access to the use of computed entry into a code table. If you only transfer a maximum of 800 pixels in sequence when copying a video line you can inline a sequence to move up to that maximum and eliminate all loop overhead in that sequence of code. You can them compute the address in the code table into which to jump or call to transfer less than 800 pixels. Chuck has abandoned the traditional **DO LOOP** constructs in Forth and replaces them with simple **BEGIN** constructs or computed jumps into unrolled inner loop code tables. Another factor at work here is the amount of code inside of the loop. The shorter the inner loop the higher the percentage of overhead for looping and the more it begs to be unrolled.

One of the most powerful techniques I have seen is what Dr. Philip Koopman documents in *STACK COMPUTERS the new wave* section 7.2.3., executable data structures. Code operates on data but they are not mutually exclusive. Code can have data embedded in it and data can be executable. This is our perspective in Forth. Some other languages and some hardware systems have a very different perspective. If you can take advantage of it it is a very powerful technique and can generate the tightest code possible on a large class of problems, problems that use decision trees.

Conclusion

Novice programmers, or commodity component programmers in some language are given a problem and start right in writing (or pasting) code. More experience programmers know that time invested in understanding the problem and designing the solution is more important than jumping into coding. The master programmer knows that this stage may deserve being done more than once or recursively. If you cobble together megabytes of code and then start testing it you would never consider starting again. You already had to produce 1000x more code than you needed to, would you want to do it again? If you carefully construct some well thought out code to match the problem at hand more thinking or a little more recoding may be more fun and more productive than the first try.

So if you're new to Forth try it out. Perhaps you can find some way to get that exhilarating 10x improvement in your programming. If you have been doing that for years remember how much fun it was when you made that improvement and consider that it could be fun again. Try for that second 10x with an open mind. OK, so you have been doing all this for years and your programs are about as small and simple and fast and clean as you think is possible. Great, but don't stop there. Unless you think you really have reached the end-all of computing be willing to try other things, experiment, look for stuff you hadn't noticed before. If on the other hand you don't care about these issues and are delightfully happy to do your sentence as a replaceable commodity programmer and you find learning painful then none of this is applicable to you. It is only for people who want smaller, faster, cleaner, clearer and better programs and more personal satisfaction.

As Chuck has said one of the rewards of thoughtful programming is the satisfaction of a job well done. If you enjoy programming you probably enjoy doing it well and you may not enjoy being told to crank out low quality code because it seems like a good idea to some bean-counting manager. It makes you wonder how many times programmers expressed concern and were forced to put Y2K bugs into programs against their will by short sighted managment looking at the quarterly budget.

One thing I am sure many people would think is that the techniques that Chuck uses are examples of individual genius and individual effort that they are not going to be applicable to typical problems in

computing. I know that Chuck would not agree with that nor do I. I have seen too many other people do it, and on teams. I worked with Chuck through my own company UltraTechnology and I worked as director of programming at the iTV Corporation with Chuck. There I was able to examine the relationship between team effort and programming environment and style. I was somewhat suprised to find that a team of Machine Forth programmers using Chuck's style of problem solving and coding combine with a management effort to guide the team worked very well. It really didn't appear that the language or style of coding had any relationship with team coordination efforts other than the relative number of bugs that different environments introduced. The Machine Forth programmers had no problem sharing problems, sharing code, and having their contributions to the effort work in harmony with the other programmers. It confirmed my experience that it is not the language but the way the team interacts that is the biggest factor. However if the coding style introduces bugs they are harder to find in a multiprogrammer project. The Machine Forth programmers consistently delivered very stable high performance bug free code on schedule. It was also my job to train these programmers on Machine Forth and I pleased to see that the techniques were easy to learn and easy to apply.

Another thing I am sure many people are going to think is that the techniques that Chuck uses is useful for dealing with small problems but not large one. However the largest and most complex, not to mention most expensive, software that I have encountered were VLSI CAD programs. I am sure people will equate the incredibly small size of OKAD and its incredibly fast code with it just being a trivial problem. Chuck's extreme optimization of the software only shows that his approach allowed him to write a suite of application programs that perform the same functions that he wanted to use in those huge expensive VLSI CAD programs. His ability to shrink a number of megabyte sized application programs into his set of a number of kilobyte sized is not an isolated case. We had a set of programmers who consistently wrote bug free 1K sized applications in Machine Forth. In one case we had one ANS Forth program that was 100K rewritten to 1K in Machine Forth and made 1000x faster by someone applying these programming techniques.

So why are Chuck's ideas about computing so unpopular? Well people tend to equate them with twenty five year old Forths rather than seeing that he has significantly changed his approach to Forth several times over the years. What is funny is that they will say they have no interest in what Chuck is doing because he is still doing what he was doing twenty years ago. Chuck is doing Forth without 'C' libraries linked in and with BLOCKS and with tiny non-ANS style Forth. Chuck would say that they are the ones who are twenty years behind and are still doing what he was doing twenty years ago, they just created a poor committee standardized version of where he was twenty years ago. So it is almost funny that both views see the other as twenty years behind.

People get very offended when Chuck just plainly says that their code is 10x bigger than required, and those are the Forth experts! The people doing 'C' or Forth in 'C' look at what he said and see those 100x numbers. They feel very insulted and sometimes claim that Chuck called them an idiot just because he reported the result of a benchmark or the time he spend coding something. They would rather refuse to believe the numbers or even refuse to look at them because they might shake up their ideas about computing. If you are selling megaForths then Chuck's ideas that Forth only needs 1K and more than that is most likely unneeded fat is not very attractive. If you are getting paid to teach people how to write ANS Forth programs you might not like Chuck advising people to try something else. If you never saw the original 10x and don't think that other people really say it you are not likely to consider other hidden potential 10x factors.

Chuck's ideas are really in the face of many of the things that are being taught in computer science. They are in the face of all the folks who's computer software has been expanding about as fast as their computer hardware for years or even sometimes falling behind. They are not applicable once you find yourself buried in self-imposed complexity. They won't help you if mix and match them too much with the conventional ideas.

While most of the Forth community has been working very hard to make Forth more like other languages to get it to fit into the niche the world has for it Chuck has been trying to go in an opposite direction. Rather than water down or dumb it down to look everything else Chuck has continued to make it smaller, simpler, faster and more productive in iteration after iteration. Just as Forth was in the face of conventional programming Chuck has chosen to not do what he thinks most everyone else is doing, making Forth look and work more like other languages. He feels that are plenty of people doing that. He wants to try to make Forth smaller, simpler, faster and more Forth-like again and again. That is in direct opposition to most of the Forth community who want to agree to set in stone the way things were done twenty years ago and then extend Forth further and further.

Neither effort seems to have done much for Forth. In this period the size of the Forth Interest Group dropped steadily, leveled for a bit, then fell off again. Today I think c.l.f is a larger Forth community than FIG. It is a shame too in the sense that FIG asks the experts to give presentations and then ask them questions. Usenet offers equal time to newbies, green belts, black belts and masters and the experts are more focused on exposing the fringes of Forth to people by endlessly debating ridiculous ways to break the standard with some of the wierdest code any of us have ever seen. I really wonder how anyone would get started with Forth today and how much of that original 10x, will be available to them.

Final Thoughts

Chuck prefers the use of seven keys, cursor and function keys rather than a mouse for precision of control. He uses a full keyboard for writing Color Forth scripts with function keys assigned as color change tokens. His GUI in OK uses the full graphics screen in a number of different modes of operation rater than having the look of popular GUI with resizable, movable, layered windows on a simulated desktop. I prefer the mouse driven interface to the seven function keys but the interface is still left, right, up, down, and a couple of buttons. I have said for years that something that looks to the user like a popular GUI with resizable, movable, layered display windows and a simulated desktop only takes a couple of K of code. Chuck has demonstrated that a windows accelerator in hardware only requires a few transistors. These things can be simple, fast, and mostly painless. The application of Chuck's methods fits well when creating a small simple GUI or OS with efficient implementations of the abstractions that the application demands.

Chuck's idea of the future is more custom silicon and machines that are efficient at what they do. All machines will not need Windows (tm) or Unix or Forth for that matter. Chuck likes the abstraction of source code being interpreted in embedded applications. He has even joked that perhaps after Y2K it will be mandated. With a tiny 1K Forth or Forth in hardware machines would embody underlying abstractions used by programmers. Chuck's methods apply well to a team of programmers setting the abstractions that they need. Chuck has said that he would enjoy seeing a small project be funded to have a small team of a few people knock off something resembling the modern GUI desktop and set

of applications in Forth on a Forth machine that does everything useful but that is a thousand times smaller and simpler than the alternatives people have now.

A tiny piece of silicon can contain hardware that performs a particular task with incredible efficiency and with a tiny general purpose central processor. One advantage of these machines will be that software will support only the devices found on chip. You can have all the special cards, video cards, analog cards, network cards, lcd support, gigabit fiber links, popular I/O interfaces etc. and for a given device there only needs to be one set of drivers to support the hardware. There is no need to support many extra layers of abstraction or inefficiency between what needs to be done, what the hardware can do, and what the software does.

After of number of years of working in these environments I must admit that I feel that I have become very spoiled. I am used to having fun solving problems quickly and feeling very productive. I have a strong sense of satisfaction with the problems that I have throught through well. I enjoy showing other people how easy it is to do things this way. I recall how when I worked as a consultant to big companies on all kinds of computer with all kinds of software that I enjoyed solving the most complex and involved problems and I think how different that world was than one where the problems are small ones and I can get so much more done. I can also understand how most people look at Chuck's methods on the surface and think that it just doesn't apply to what they have to do. Chuck's methods wouldn't solve all of the problems that they have. Their problems are often related to different methods and Chuck strategy is to avoid most of the problems other people must face to be more effective at solving the problems he wants to solve.

Forth Essay Chapter 1

Forth Essay Chapter 2

Related references at this site

Fireside Chat 2000 **Chuck Moore** to SVFIG 11/11/00
Fireside Chat 1999 **Chuck Moore** to SVFIG 12/18/99
1x Forth **Chuck Moore** 4/13/99
Chuck Moore interviewed in his home 6/6/93
Fireside Chat 1998 **Charles Moore** 11/30/98 (first 50 min)
Color Forth (posted 11/2/00)
Dispelling the User Illusion w/ **Color Forth** code examples. **Chuck Moore** to SVFig on 5/22/99
OKAD reference page 9/3/00
Forth Chip Reference Page Jeff Fox 8/2000
People at UltraTechnology Chuck Moore
Chuck Moore (owner of Computer Cowboys)
More on Forth Engines Volume 16, 1992 - **OKAD articles** **Charles Moore**, Dr. C. H. Ting, 2/25/99
Color Forth Update 9/16/97
Color Forth **Charles Moore** 7/27/97
Fireside Chat 1996 **Charles Moore** 11/16/96
Life Beyond MuP21 **Charles Moore** to the SVFIG 5/27/95
MuP21 a High Performance MISC Processor Dr. C.H. Ting and Charles Moore 3/17/95

Chuck Moore to SVFIG on 4/23/93.
Fireside Chat 1993 **Charles Moore** to SVFIG 11/93
Forth - a Language for Interactive Computing **Charles Moore 1970 HTML** (first Forth document), posted 1995
PDF version (formatted like original typed paper) posted 6/2000
Microsoft Word version (formatted like original typed paper) posted 1995
UltraTechnology Homepage
Forth (Thoughtful Programming) Jeff Fox 12/99
Introducing Aha
Aha Jeff Fox 11/29/00
F21 in a mouse GUI demo of desktop with application in 600 words of code. 3/8/00
UltraTechnology site dated index with info on and pictures of F21d tests.
Streaming Video Theater
Forth -- the LEGO of Programming Languages 1/19/95 11/00
Parallel Forth - the new approach **OCCAM and Forth-Linda, Dr. M. Montvelishsky, FD 1993**
F21 and F*F : **F21 and Parallelizing Forth**, J. Fox, Dr. M. Montvelishsky, FORML 1993
Low Fat Computing Jeff Fox 12/6/98
ANSI Forth is ANTI Forth Jeff Fox 2/28/99, **Chuck Moore** 7/26/98, 3/5/99
Distributed Shared Memory in Forth **Parallel Forth**, J. Fox, 1995
Forth-Linda **Parallelizing Forth**, J. Fox, **FORML 1991**
Forth Stamp Jeff Fox 9/2/00
Forth Meta Compilation J. Fox 8/21/97
Forth Meets *Laws of Form* 1994 J. Fox
F21 Chess 1.6 8/12/97
html guide to **OK** version 1.01 Source code to **OK** ver 1.01 for the MuP21 4/18/95
P21Forth 1.02 User's Manual in hypertext and Word for Windows formats 4/16/95