$10 Volume XVII, Number 2 July 1995 August

FORTH DIMENSIONS

VLSI Design

Forth in Control

Compiling ANS Forth

Pinhole Optimization

Code Size, Abstraction, & Factoring

Contents

Features

Departments

Editorial

Guilding the Lily

Amid the finger pointing and hair splitting over why Forth hasn't set the prairies on fire, we often neglect steps we can take to improve the situation. Sure, it's fun to find scapegoats ("poor implementations cause bad first impressions" and "Forth-83 sank the ship"), to devise rationalizations ("C had Bell Labs to push it" and "Forth is too creative for the grunt-code production world"), and to take technical detours ("marketing Forth is too hard; let's talk about loops, threading, vocabularies..."). I've opined about the need for Forth vendors to do cooperative marketing, and about things the Forth Interest Group can do differently or better. It's only fair to consider how the programmers can promote Forth's acceptance and improve their ability to get Forth work.

Forth programmers should form a professional Guild along the lines of what skilled artisans have done throughout much of modern history. It need not be totalitarian, but—through the soundness of its organizing principles—it should wield enough clout to achieve industry recognition, to lend authority to its endorsements, to make admission into it desirable, and to require of its members adherence to a formal "standard of practice."

Okay, so organizing a body of anarchistic Forth programmers might not be the easy path, and it might dilute our resources. But the same can be said of aerobic activity, and that's *good* for you. My point: this exercise, too, might prove useful.

Achieving support for, and cohesion within, a Forth Programmers Guild will require defining its mission carefully. Following are a few agenda items that should be addressed in the Guild's standard of practice:

- *Certification of Forth programmers.* A way to verify someone's Forth expertise will be a boon to would-be employers, and will give professional accreditation to those seeking Forth work—perhaps on a graduated scale. It will also minimize the damage to Forth's reputation caused by incompetence, self-indulgence, or poor judgment.
- *Certification of training programs.* Guild-endorsed classes and workshops (whether sponsored by vendors, independents, or the Guild itself) will provide a clear path, with specific objectives, for anyone desiring to learn Forth or to improve their level of expertise.
- *Good style has substance.* Every Guild member's code will demonstrate Forth's readability. This will improve code maintainability, will mitigate the impression that everyone uses Forth differently, and will help non-Forth managers and executives who have to make sense of (or at least look at) Forth code. To this end, a common Forth coding style, or one of several Guild style conventions, will be followed.
- *Professional conduct.* Ethical business practices regarding deliverables, documentation, accountability, honest representations, copyright, etc. should be a matter of course but should not be taken for granted; making these part of Guild members' formal obligations will be appreciated by employers and will underscore the integrity of the Guild.
- *Support the Forth economy.* Commercial Forths will be used for all new commercial projects, and will be recommended whenever working on existing products based on non-standard, obsolete, and/or in-house dialects. This will build a stronger vendor base, make clients' code easier to maintain after the original programmer leaves, facilitate the training of additional or replacement programmers, focus resources on the application (rather than on rolling another Forth), and strengthen Forth's professional image. It leaves unsupported public-domain Forths to amateur, hobbyist, and experimental pursuits.
- *Dissemination of knowledge.* Forth has much of value to offer, even to users of other languages. Guild members should be encouraged to speak and write about their application of its methods and philosophy, as well as to help others learn Forth.

Too much controversy will undermine a fledgling Guild and, like any new organization, it should fill a specific, unmet need and use its own particular strengths, while avoiding dilution of purpose. I recommend that the Guild stringently refrain from any activity or overhead which might drain its resources or soften its focus. Its founders might use the umbrella of the Forth Interest Group as its infrastructure.

Concrete steps like these will enable Forth users to improve their lot instead of getting mired in debates and conjecture about circumstances over which they have little control.

—Marlin Ouverson, editor
FDeditor@aol.com

Forth Dimensions
Volume XVII, Number 2
July 1995 August

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

Forth Dimensions welcomes editorial material, letters to the editor, and comments from its readers. No responsibility is assumed for accuracy of submissions.

Subscription to *Forth Dimensions* is included with membership in the Forth Interest Group at $40 per year ($52 overseas air). For membership, change of address, and to submit items for publication, the address is: Forth Interest Group, P.O. Box 2154, Oakland, California 94621. Administrative offices: 510-89-FORTH. Fax: 510-535-1295. Advertising sales: 805-946-2272.

The Forth Interest Group

The Forth Interest Group is the association of programmers, managers, and engineers who create practical, Forth-based solutions to real-world needs. Many research hardware and software designs that will advance the general state of the art. FIG provides a climate of intellectual exchange and benefits intended to assist each of its members. Publications, conferences, seminars, telecommunications, and area chapter meetings are among its activities.

"*Forth Dimensions* (ISSN 0884-0822) is published bimonthly for $40/46/52 per year by the Forth Interest Group, 4800 Allendale Ave., Oakland, CA 94619. Second-class postage paid at Oakland, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621-0054."

Letters

No Better Course

Dear Marlin,

I empathize with Richard Fothergill who, in his letter published in the March/April issue, asked whether *Forth Dimensions* were suitable for a beginning Forth programmer.

Argumentation, Exposition, or Inspiration?

I myself began learning Forth while the CASE statement insanity was in full bloom, and I found it rather difficult to distinguish the stuff of substance from the inane. Issue after issue of *Forth Dimensions* devoted space to varied implementations of CASE, a word of which I have never yet made use. I, too, was intimidated because I could not see why so much attention was being devoted to such an apparently arcane matter. Looking back, I am reminded of the faerie tale of the emperor's new clothes. A newcomer to any field, including that of Forth, does well to seek out someone having a bit of experience who can provide perspective.

As is the case with most magazines, the coverage of *Forth Dimensions* is basically limited to the topics of articles which are submitted for publication. However, articles need not be directed to basic programming techniques to be of interest to beginners. Indeed, it would be very difficult to surpass the coverage of such techniques provided by *Starting Forth*; to make the attempt in *Forth Dimensions* would be to squander the available space.

In an egoless programming environment, every member of the team is encouraged to interact.

Tyro and expert alike should take interest in the challenge of finding find better ways to approach difficult applications. There suddenly is a great deal of interest in development software for embedded systems. It seems that magazine articles are increasingly containing admissions by programmers that the C language is unsatisfactory for the task. Few outside the Forth community seem aware of umbilical compilation, a powerful and economical approach to the programming of embedded systems. The approach appears to have been developed using Forth. Since the world is, at this moment, actively searching for a better solution to the programming of embedded systems, I should think each issue of *Forth Dimensions* would be full of articles dealing with the subject. I would expect to find perhaps an occasional article describing a complete system, but I would expect to find many more articles discussing the problems encountered, viable solutions, and the considerations involved. In short, I would expect to find mainly articles which provide a stimulus for refinement and innovation. Thus, I see great potential value in *Forth Dimensions* as a forum for the synthesis of solutions to current application needs.

The monumental tome by Richard Rhodes, *The Making of the Atomic Bomb*, provides insight into the workings of nineteenth-century science, particularly in the fields of physics and chemistry. The period was one of dizzying advance, with discovery after discovery, each following hard upon the heels of the last. The predominant characteristic of the era seems to have been the almost completely unfettered interaction between the scientific minds of the age. It appears that this interaction was the catalyst for the resultant growth of knowledge, a growth which proved explosive in more ways than one. Not until most of the foundations of atomic theory were in place did the governments of the world seek to curtail this interaction. Curiously, patents were one means of suppression. One cannot read the book without being struck by the great power of interacting minds, and, at the same time, by the blindness and mind-sets to which minds deprived of interaction are prone.

Foundations

Regarding *Starting Forth*, I can recommend to the beginner no better course in Forth technique and philosophy. However, the book is deceptively simple. I dare say that only a small portion of those proffering their services as Forth programmers have actually mastered all the techniques covered therein. One way to ensure such mastery is to proceed through the book *while sitting in front of a computer*, working out each and every exercise. It is very important to stop and get help whenever a topic or technique is not clear, for the book contains little material which is not significant. Beyond *Starting Forth*, the best one can do to educate himself in Forth is to dissect and comment well-written Forth code, such as that of a first-class development environment or an application written by someone who has truly mastered Forth. Much of the Forth code I have seen, including that of some commercial Forth systems, can only be described as a "dirty hack," being confusing and even ugly to the eye. I think there are few programmers who truly understand the philosophy of the language, and fewer still who can implement that philosophy.

To Be or Not to Be

I strongly recommend to Richard, and to the Forth community as a whole, a technique espoused by Gerald M. Weinberg in his classic book *The Psychology of Computer Programming* (Computer Science Series, Van Nostrand Reinhold, 1971). (Those without a personal copy of the book should make it a high priority to get one.)

Weinberg endorses "egoless" programming, a practice in which each member of a programming team routinely submits his work for inspection by other members of the group. The submission is not a formal, structured affair; indeed, the less formal and less structured, the better. Perhaps the best mode of submission is simply to hand a listing to an associate and say, "Here; take a look at this and tell me what you think."

In an egoless programming environment, every member of the team is encouraged to interact: to read programs written by other team members, to ask questions, to comment, and to share responsibility for the productivity of the team. Everyone on the team should be familiar with what everyone else is doing. Accordingly, schedules and critical paths should be relatively easy to estimate. Since no team member "owns" a particular piece of code, the workload can be redistributed as necessary, with minimal risk of offending a team member. Loss of a team member need not be catastrophic. No team member need fear struggling alone with a problem beyond his expertise, since egoless programming encourages team members to ask for help and attaches no stigma to ignorance, unless the ignorance goes unremedied. Program maintenance is facilitated, since there is little chance of incorporating into the program an obscure or exotic technique which only one programmer understands. Not that such techniques should be avoided; rather, when employed, they should be made thoroughly clear to the whole team. The egoless programming team is an excellent environment for a novice: it provides him opportunity to expand and mature his skills. The team is also an excellent environment for the expert: it provides him opportunity to teach and direct. Of course, not everyone functions well in such an environment.

The Virtue of Reading

Egoless programming need not be restricted to the activity of a programming team. Even a programmer working alone may put the concept into practice: he need only seek out one or more associates with whom he may, on a habitual and frequent basis, exchange and discuss listings. The epitome of such interaction was found in the days of batch processing in a mainframe environment. Programmers waiting for their runs, having nothing better to do, would read one another's programs, and would informally discuss things they found interesting. Indeed, the cornerstone of egoless programming is the *reading* of programs. Too frequently, the only entity (other than the programmer) which reads a program is the computer. Only by reading a program do some aspects of its nature become apparent. Is the program lucid? logical? clean? elegant? The only way to tell is to read it, but the programmer who wrote it is hardly an objective evaluator.

Hangin' Out

Ironically, the advent of the personal computer, while increasing the interactivity between man and machine, has significantly *decreased* the interactivity between programmers, even when there are a number of programmers in a "shop." The isolation can become almost total when a programmer single-handedly undertakes a project for a client who fears that "proprietary" techniques may be leaked to his competitors.

The best bet I see is to rendezvous with local Forth programmers as frequently as possible. Try to find a pizza parlor or a low-key cafe (a place with good lighting and no jukebox) where you can comfortably spend an hour or so in one-on-one conversation. A daily meeting over lunch or right after work would be terrific, but a weekly meeting may be more realistic. The only reason for having a schedule of any sort is to increase the probability of encounters, particularly for people who have to drive across town to meet. Get into the habit of carrying around a listing of the program on which you are currently working. A few words of caution: regimentation is a sure way to stifle interest. *Don't* try to turn a relaxed, informal gathering into a group presentation; the goal is one-on-one interactivity, not a lecture. *Don't* establish official start and end times, insisting that everyone act in concert; encourage individuals to come and go as their needs and interests dictate. Just get into the habit of getting together, swapping listings, and talking about the code. Everyone will benefit.

Regards,
Russell L. Harris
8609 Cedardale Drive
Houston, Texas 77055–4806
713–461–1618
713–461–0081 (fax)

Compiling ANS Forth

Tom Almy
Tualatin, Oregon

Introduction

ForthCMP is the author's native code metacompiler for Forth. In 1982 I wrote NCC, which was the first Native Code Compiler for interactive Forth environments. A year later, the compiler was ported to LMI Forth and is still sold as part of the LMI Forth package. The NCC exists in versions for Z80, 80x86, and 80386 protected mode.

However, I felt that for optimum performance a metacompiler was necessary. A fully compiled Forth application would be faster and more compact than one tied to an interpretive environment. So ForthCMP (then called Cforth) was born. ForthCMP is run from the DOS command line to compile Forth applications directly into executable machine code files. ForthCMP-generated, executable files are much smaller than comparable executables produced by other language compilers, and are typically smaller than traditionally metacompiled Forth applications. Execution speed is comparable to current C compilers.

Since ForthCMP is itself written in Forth, application programs can evaluate expressions and execute application colon definitions during compilation time (these are called 'host colon definitions'). Variables and other data structures exist in the target (application) address space, but can be accessed during compilation as though in the host. Thus, data structures can be algorithmically initialized during compilation. Code definitions are allowed, and generate target words and machine code subroutines. The major difference between ForthCMP and a traditional interpreted Forth is in the generation of colon definitions. In ForthCMP, they are machine code subroutines. The system stack is used as the data stack, and a separate stack is used for the return stack. Several calling conventions are used to handle the subroutine return address, depending on usage hints provided to the compiler. The usage hints can allow passing of arguments in registers and eliminate the need to move the return address between the data and return stacks.

The Compilation Process

In a traditional Forth, the compiler treats words in three classes. Words marked as `IMMEDIATE` are executed during compilation, and generally represent control structures. Words not so marked are compiled and then executed when the word is executed. Words not in the dictionary must be numeric literals, which are compiled as inline constants.

In ForthCMP, there are many more classes of words:

- Literals—these are not compiled, but are pushed on a compilation 'literal stack.' This allows evaluation of literal subexpressions at compile time, and also allows generating instructions with literal arguments, which reduces the amount of code generated.
- Intrinsics—these words are built into the compiler and are like traditional `IMMEDIATE` words; however, most primitive, standard Forth words are intrinsics so that they can generate inline code. The ANS Forth version of ForthCMP has about 120 intrinsics. Application programs cannot define intrinsics (which means they cannot define compiling, immediate words). Intrinsics have no execution tokens, since they compile inline code.
- Functions—colon and code definitions, either part of the application or compiled when needed from a library. It is possible to forward reference functions, with the restriction that the execution token is not directly accessible until the function is defined. Words defined for the host environment cannot be referenced in the target. Only functions have execution tokens—the value returned by `'` (tick) or `FIND`—that are suitable for `EXECUTE`, and then only if the `IN/OUT` compiler hint (which allows passing arguments in registers) is not used.
- Constants—words defined as `CONSTANT` or `2CONSTANT` behave as numeric literals when compiled. Constants are not allocated any memory in the target image, thus have no data address or execution token.
- Variables—words defined as `VARIABLE`, `2VARIABLE`, `SCONSTANT`, or `CREATE` behave as numeric literals when compiled. There is no code generated, and the execution token is the data address.
- Arrays and Tables—ForthCMP provides defining words to generate single-dimensional arrays and tables (constant arrays). These generate inline accessing code when used in a colon definition. The execution token is the data address.

- Value—words defined as VALUE generate inline accessing code when used in a colon definition. The execution token is the data address.
- Does words—created by executing host words containing DOES> or ;CODE. These compile a numeric literal (the data address) followed by code to invoke the does code. The execution token is the data address.

The Challenges

Converting to ANS Forth simplified many aspects of the compiler, but added a number of challenges. These challenges, to be discussed in this article, are:

- Lack of application 'compiling words.'
- UNLOOP—an easy function for interpreters, but a terror for a compiler.
- Execution tokens—they don't always exist.
- UNUSED—how to calculate available memory in a compiler environment.

Compiling Words

Forth has traditionally allowed writing words which alter the compilation of other words. This is, perhaps, one of the strongest features of the language. However, these 'compiling words' tend not to be portable among implementations. ANS Forth attempts to solve the portability problem by providing new, higher-level, portable functions, and by restricting the way these functions can be used. However, in the compilation environment, these functions would have to be restricted further. To write a true compiling word for ForthCMP is a difficult task, so it has been disallowed! For instance, consider the mundane word, BEGIN. In a typical interpreter, it would be written as:

```
: BEGIN
  AHEAD \ compile forward reference
  2     \ token for syntax error detection
; IMMEDIATE
```

While in ForthCMP it is:

```
: BEGIN  FT     \ go to Forth vocabulary
  FLUSHPOOLS    \ flush literal stack,
                \ but not cpu registers
  NOCC          \ don't rely on any set
                \ condition codes
  ASM BEGIN, FT   \ equivalent to "AHEAD"
  CSTATUS       \ preserve current
                \ register usage so it
                \ can be restored on
                \ UNTIL or REPEAT
  2             \ token for syntax error
                \ detection
;
```

Optimization gets in the way of clean implementation. Those nice, simple Forth compiling words just can't be implemented simply in ForthCMP.

Without the ability to generate immediate words, many standard words lose their usefulness. For instance, LITERAL has two uses. The first, when used in an immediate word, is to compile a literal into the word being defined. As mentioned before, literals are not compiled, so even if user immediate words were allowed, this word could not perform as desired. The second use of LITERAL is to evaluate expressions at compile time (e.g. ' [FOO 3 CELLS +] LITERAL). However, all ForthCMP literal expressions are evaluated at compile time. In fact, the [and] words have been removed, since they have no usage!

The Terror Of Unloop

UNLOOP is defined to 'discard the loop control parameters for the current nesting level.' Typical implementation in an interpreter is a non-immediate word that drops words from the return stack. This allows executing EXIT from within a LOOP.

In a compiler, however, the 'loop control parameters' can vary in number depending on the loop usage, and might not be on the stack. While the compiler could easily generate what unlooping code was necessary in order to do an EXIT, something which is difficult for an interpreter, it cannot readily separate the functionality of UNLOOP and EXIT. Consider the following word:

```
: DUMMY
  10 0 DO   10 0 DO  X DUP
  IF  UNLOOP
  THEN  I .
  IF UNLOOP EXIT
  THEN  LOOP  LOOP ;
```

If X returns a false value, the inner loop index is printed. If X returns a true value, the first UNLOOP will cause the outer loop index to be printed, and then the second UNLOOP and EXIT will cause the function to be exited. However, since the generated code is static, the code to implement I must be the same for both the inner and outer loops. This cannot be guaranteed except by disabling any optimization whenever UNLOOP occurs inside the loop. This is a highly undesirable situation!

A lesser problem is that UNLOOP is basically a control structure word, but without any compile-time error checking. In order to execute correctly, a correct number of UNLOOPs must precede an EXIT. To ensure correct structure, and to solve the compilation problem, UNLOOP was considered to be part of a new control structure, UNLOOP...EXIT. Now compile-time syntax checking can prevent unstructured use of UNLOOP (as in the example) and make certain that the correct number of UNLOOPs are used. In the example above, when the first THEN is reached, the compiler complains that EXIT is missing, because the UNLOOP...EXIT control structure is incomplete.

Now the code generation task is straightforward. ForthCMP has a compilation loop stack which keeps track of loop structures during compilation. Normally, DO (and ?DO) push information on the loop stack that is queried by I (and J) and removed by LOOP (and +LOOP). A second stack pointer, the 'unloop pointer' was added that is set by the DO and LOOP words to match the loop stack pointer. This unloop pointer is used by I to query loop information. The UNLOOP word sets the unloop pointer back so

that any loop index referencing that follows will access the proper data. EXIT restores the unloop pointer to equal the loop stack pointer. Of course, UNLOOP also generates code to remove any runtime loop parameters. The final UNLOOP code is [in Figure One].

The Elusive Execution Token

Earlier Forth standards involved themselves with implementation details by referring to the code, link, name, and parameter fields of words. ANS Forth, thankfully, tries to avoid the implementation details. It basically states that words like ' and FIND return an 'execution token.' The execution token can only be used in limited ways. It can be used as the argument to EXECUTE or COMPILE, (the latter, since it is only useful in compiling words, is not found in ForthCMP). If the execution token comes from a word generated with CREATE, it can be used as an argument to >BODY to get the data address.

ForthCMP exceeds the standard as far as the data address is concerned, since >BODY can be used with the execution token of words generated with CREATE or VARIABLE (among others). And as a non-portable bonus, >BODY is a no-op; the execution token is the data address. However, ForthCMP's execution tokens can't generally be passed to EXECUTE.

In an interpreter, all words have code associated with them. While in ForthCMP, only colon definitions and code words have code, all others generating either literal values or compiling inline referencing code. It doesn't make sense to waste memory with code segments that would only get executed if EXECUTE were used.

However, there is a way around the problem: any word for which one wants to have a real execution token can be embedded in a colon definition (or code word), which does have an execution token. This is the same technique traditionally used to put literals in execution vector tables. Consider this short example:

```
10 CONSTANT A       \ Constants don't have
                    \ execution tokens

PRIMITIVE           \ Generate a better
                    \ calling convention
: EXEC-A  A ;       \ Colon definitions
                    \ have execution tokens!

VARIABLE EXECV      \ variable we place
                    \ execution token into
' EXEC-A EXECV !

: MAIN              \ Main function fetches
                    \ execution token,
                    \ executes it
                    \ and prints its value.
EXECV @ EXECUTE .
;
```

The code generated by EXEC-A:

```
POP     SI          \ return address
PUSH    000A        \ constant value
JMP     SI          \ does return
```

And for MAIN:

```
MOV     AX,[012B]   \ Fetch contents of
                    \ EXECV
CALL    AX          \ call function
POP     AX          \ move return value
                    \ from stack to
                    \ register
JMP     0136        \ jump to ., which does
                    \ return from MAIN
```

Used By Unused

This is simply the amount of remaining space in the region starting at HERE. The assumption was made that,

Figure One. Final version of UNLOOP.

```
: UNLOOP FT     \ FT changes the vocabulary to FORTH from cross-compiler
INDL? IF        \ INDL? returns true if inside a DO LOOP
ULPTR  @ LUPSTK = IF                  \ Unloop pointer at start of stack?
WARN" too many unloops" EXIT THEN
ULPTR @ CELL- @                       \ check top of unloop stack
DUP 0= IF  ASM BP INC BP INC FT THEN \ pop stack at runtime
1 = IF -8 ELSE -6 THEN ULPTR +!       \ pop unloop stack now
?RESCX          \ generate code to restore loop index register, if used
THEN

\ 0A on the data stack top during compilation is used to indicate being
\ within an UNLOOP..EXIT control structure. So put a 0A on the stack
\ top unless there already is one present.

DEPTH IF DUP 0A <> IF 0A THEN ELSE 0A THEN
;
```

Figure Two. Conditional compilation tailored to memory model.

```
UNDEF UNUSED            \ compile what follows only if UNUSED is needed
                        \ to resolve a reference.

CODE UNUSED  SI POP     \ stash return address
SEPSSEG? 0= [IF]
SP AX MOV               \ if stack segment is same as data segment, then
                        \ top of memory is at stack pointer location.
[ELSE] SEPDSEG? [IF]
dssize 10 * # AX MOV    \ else if data segment is different from
                        \ code segment,
                        \ then top of memory is top of allocated segment
                        \ (paragraph count)
[ELSE]                  \ Code and data segments the same, stack segment
                        \ different.
FIND PSIZE [IF]
DROP PSIZE              \ program size defined -- use it
[ELSE]
FFFE                    \ single segment and program size not defined.
[THEN]
# AX MOV
[THEN] [THEN]
DP [] AX SUB            \ subtract dictionary pointer
AX PUSH                 \ return value on stack
SI JMPI                 \ return from function
END_CODE
[THEN]                  \ ends the UNDEF
```


during compilation, this value should be the space remaining in the target image, assuming nothing else to be compiled. The situation is complicated by all the possible memory models that can change these values. During compilation, the value is estimated by subtracting the target value HERE from the size of the target data segment (normally 64K, but can be less). At runtime, the library has a definition of UNUSED that provides an exact value. The definition makes use of conditional compilation to compile code tailored for the particular memory model [see Figure Two].

This generates a five-instruction subroutine for any memory model. The process is certainly more complicated than one might expect!

Conclusion

The new ANS Forth standard provided some unexpected challenges which could not be completely solved. However, the situation has improved over earlier standards. I expect that, as Forth compilation is more widely used, future standards will reflect these approaches.

Tom Almy has been a user of Forth since 1981, when he used it to implement an integrated circuit layout program, PRIDE. Graduating from Stanford with an MSEE and hired by Tektronix in 1973 to design processors, Tom joined Tektronix's research laboratories in 1979, where he has since been involved with various aspects of integrated circuit and system design. Outside the work environment, Tom enjoys hiking and running a literary bulletin board system. He can be reached at tom.almy@tek.com via e-mail.

A Forth Story

Allen Cekorich
Walnut Creek, California

In 1975, I was fresh out of college with a bachelors degree in physics and a need to find a job that paid enough to support myself. After a six-month search, I landed an entry-level position at MDH Industries in Monrovia, California, where they promised to increase my minimum wage starting salary once I proved myself useful. I was to work with the president and owner, Dr. Howard Marshall, Ph.D., on a project to develop an instrument to measure the real-time sulfur content of a coal stream by detecting and analyzing the prompt gamma ray spectrum resulting from the absorption of thermal neutrons from a Californium 252 radioactive source. The opportunity sounded exciting to my naive ears, as it was my first professional job. I was finally in the real work world of adults and ready to go on to do great things. Remember, this was twenty years ago and I was only twenty-two years old. I had no previous experience to guide me in the situation I was in, no mentor to teach me, no helping hand and no idea of where to begin or what to do. Like most good engineers, I decided to fake it until I understood my value.

I told him the final device could not be programmed in Forth. Why would I say such a thing?

I spent the first year or so understanding the design parameters of a sulfur meter, which involved creating a computer model on a Tymshare IBM 370 system accessed with a teletype terminal at the fantastic rate of 30 characters per second. This was a revolution from the punch cards I had used in college on a CDC 3150 that could be viewed through a glass window in the foyer of the computer department. My degree in physics mandated that the modeling language would be Fortran, and I naturally enjoyed programming, which was probably related to my ability and love for mathematics. I had completed the coursework for a degree in mathematics with a 4.0 average in college. I was proud of the growing complexity and depth of my simulation, which was now consuming hours of computer time during the night hours when the cost was lowest.

It came to pass by the natural events of the development process that construction of a sulfur meter prototype was to take place. Howard Marshall had earned his doctorate in physics from the California Institute of Technology, which is very much in bed with the Jet Propulsion Laboratory in Pasadena. His contacts encouraged a computer-based instrument and recommended a strange language called Forth that was floating around the lab. They convinced him it was possible to build a small, Z80-based controller that could acquire the spectrum from the sodium iodide detector and maintain real-time signal-processing rates to compute the actual sulfur content. Somehow, outside my area of responsibility, an S100-bus system showed up from the Empirical Research Group. My electrical engineer office mate had been assigned the task of fooling around with it for development purposes, but no progress seemed to ever be at hand. After some time had passed, a fellow named Martin Smith showed up from the Cal Tech network. He brought with him a Forth running on a Monolithic Z80-based multibus system, and progress toward a controller began.

I was preoccupied with designing the sulfur meter based on my Fortran simulation, but the natural need to verify the model from real data taken from the prototype was growing important. With the help of Marty, I started playing with the Forth computer. This was the first time in my life that I had actual, physical contact with a computer. Those big, eight-inch Shugart floppy drives that held a whopping 120K bytes, and the awesome 64K of fast RAM, along with the character-based video display terminal, intoxicated me. But what was more puzzling was this strange way of talking to the computer, called Forth. I had taken a computer language survey class in college which included Fortran, Algol, PL/M, Cobol, Trac, Lisp, and APL, but had never heard of anything called Forth. It was strange and unnatural without program statements. I could not find the compiler, linker, or listing output. I could not figure out how it worked, but I realized that I now had direct contact with the CPU without the need to learn complex system manuals. I finally had a computer under my control and I went to town. Over the next few years, I had a good time writing programs on that small Forth system to do data

preprocessing for input to the real, grown-up IBM Tymshare computer for comparison to my simulation.

I taught myself the Z80 assembler in the Forth, which gave me access to the computer hardware. I played with serial ports, DMA for the disk drive, video display controller, interrupts, and, most important of all, the AMD 9511 floating-point coprocessor. I wrote linear matrix functions, least squares routines, statistical measures, data filters, and data stacks for working with the gamma ray spectra. I used that little 64K computer to its limit to complete the calibration of the first delivered sulfur meter. I also became an expert in using Forth, although I still did not fully understand what I was doing.

About this time, around the beginning of the eighties, a recruiter called me searching for a Forth programmer. I was not a Forth programmer in my mind. I did not see myself as a Forth programmer. I was a physicist using a tool to do a job. Anyway, I went on an interview to Forth, Inc. in Manhattan Beach, California, and met a man named Edward Conklin. We talked about what I had been doing with Forth, I showed him some of my listings, toured the offices, and shook hands upon departing. A few days later, the recruiter called saying I had been offered a job for more money than I was making and encouraged me to accept. I was puzzled. I was not a Forth programmer. Why did they want me? What would it be like? I just did not understand where I would fit in. I declined the position. Over the years, I have wondered what direction my career would have taken if I had just said yes. Looking back, it is easy to see now that I was an exceptional Forth programmer, as following parts of my story will reveal; but remember, I was still in my twenties, working on my first job, which limited my view of my career horizon. My destiny would take me to other places and back again, to Forth.

My job was winding up with the completion of the first sulfur meter. Martin Smith had left earlier, going back to a previous job at Technicolor in Hollywood. I had grown as an engineer, becoming more than could be supported by the twenty-five person company that was owned by a single person. The end of my first job had come, so I resigned and, that weekend, I bought a Victor 9000 personal computer. I did have the vision to see that what I had done with that small Z80 computer spelled the death of my cherished IBM 370 computer running Fortran over the phone line. The future was in small, personal systems that clearly exceeded the power of the off-line dinosaurs. I did not know what I would be doing, but I knew what I would have to do and that was to learn the basics of the smaller machines. As fate would have it, Marty called me the following Monday, and a week later I was working for Technicolor. It was now May of 1983.

I had taken the job as a computer programmer who would develop the control software for a film printer. This was a special kind of printer to reduce the cost of choosing the RGB color light levels for a film scene by printing one frame of scene onto the positive, thereby saving the cost of developing a complete print just to check the production parameters. I had to learn Intel's version of PL/M and assembly for the 8086, which was the heart of the Intel 88/40 multibus board computer controller. I was back to compilers, linkers, and locators, and got to play with in-circuit emulators. I discovered real-time control that enables computers to interact with the physical world. I learned multitasking, interrupt response times, and control inputs and outputs from A/D and D/A ships, counters, and parallel lines. I got to play with big servo motors. I had a ball. I almost forgot about Forth.

But not completely. I obtained the fig-Forth listing for the 8086 with the intention to boot a Forth on my Victor 9000. I spent many nights and weekends typing in the listing as I learned about DOS and just how the PC worked. I gathered documentation, bought some software, and joined the Victor 9000 users group. Time passed. Work went well with the nearing completion of the proof printer. Then Hollywood came into the picture. Hollywood is not in the real world, they make it up as they like. The engineering director and the chief engineer got themselves fired for no apparent reason other than a pompous power play which I never understood. The upshot was that my project was canceled, leaving me in limbo. I chose to resign a month later, simply because I no longer had a defined job. It was July of 1984.

I spent the next five months working twelve-hour days at home on my Forth system. I booted the kernel as a DOS executable, and promptly rewrote it to my liking, including redesigning the inner interpreter. I was forced, by finally understanding the true nature of Forth, to add an 8086 assembler of original design, an interface to DOS for files and display control, floating-point words for an 8087 coprocessor, and many utilities to work with my system. Looking back, I wonder why I did it. Why would I create a complete Forth development system for the PC? I had no use for it, nor any future plans for the system. I believe the answer was just to learn how personal computers work. Forth gave me direct access to the machine, and freedom to explore designs that felt right to me. I did not have to depend on a vendor to show me what I could do. My physics training led me to explore fundamentals of computer operation much as I explored the fundamental laws of the physical world. I also began reading *BYTE* magazine every month to keep up on the technology, and I read books such as Donald Knuth's *Art of Computer Programming*. Forth gave me freedom to explore beyond the limitations of a fixed compiler with a straight-jacket syntax. I had finally caught the Forth bug.

The realities of living without an income finally caught up with me. In December of 1984, I found a job with Litton Industries. The fiber optic department under Mike Suman had a contract with Western Geophysical to build a demonstration underwater fiber optic microphone towed array for oil exploration. The job was to complete an Intel multibus computer demodulator for five sensors. The software was written in PL/M-86 and assembler, and was presented to me as almost done. I learned quickly that the statement was politically correct but entirely false from an engineering perspective. I had wandered blindly into defense engineering for the first time. I redesigned the computer system to use three single-board computers in

a multibus backplane, and wrote the software from scratch to multiplex the three 8086 CPUs to accomplish the demodulation. Four months later, it was finished, complete with a real-time video display of the sensor outputs for the all-important demo. The next day, the contract was canceled due to the oil glut of the mid-eighties.

I wondered if I had a job. The major part of the fiber optic directorate was involved in fiber optic rotation sensor development for military applications. The program seemed to be headed by a man named Dr. George Pavlath, who was a Ph.D. in physics from Stanford University. He had a problem with the testing of the rotation sensors on the rate tables which used H-P BASIC controllers. He knew from my resume that I had experience with Forth, and he had heard from his friends at Stanford that it was a very neat control language. I told him I had developed my own Forth system for the PC, and we agreed to try it out for rate table control. I brought in my system and spent a few months porting it to the IBM PC, adding drivers to read IEEE-488 instruments via a Metrabyte card, and rate table position from a CTM-05 counter board. I completed a fully automated rate table test station and began to test fiber optic gyros.

The application of Forth to a flexible test environment was perfect. I went much further and added online data analysis, and began constructing my own unique plotting additions to Forth based on Brodie's MAKE DOER construct. My Forth system grew to maturity as different problems demanded solutions. I quickly returned to my physics roots to contribute to the new technology of fiber optic sensor development.

All was not well, though. I encountered the first resistance to the Forth way of doing business. The Forth development environment was so efficient that the H-P BASIC controllers were made obsolete. This led to resentment by engineers who had invested their time learning H-P BASIC. I offered to teach them Forth, but remember, this was a system I had personally written. It was my creation, and no one could understand it as well as myself. Why should they invest part of their careers learning a tool that was the personal property of one engineer? They did have a point. But the fact was that my system was doing things that nobody thought possible before. It was even worse than that. It turned out that someone in a different Litton division was using Forth for production test station control for the same reason, its efficiency and power. This person was upset that I had brought in a new dialect. He had his box of tools and would not look at mine, and we could not share code.

As the years passed, my system became entrenched in the fiber optic directorate and enabled a lot of progress to be made, even though I spent most of my time concentrating on the physics of the devices. A few people finally learned my system, which became quite powerful, but the lingering resentment remained. Other engineers felt I had cheated by using Forth, that it was unfair to them. I even published my Forth system, called AFORTH, through the Victor 9000 users group. I was told that up to forty people bought copies, and the head of the users group eventually got a job writing Forth.

Sometime in the eighties, I got it in my head that Forth could be applied to build fast demodulators, especially since the advent of Forth chips. I convinced George Pavlath to send me to a company called Novix to check out a Forth-based CPU. It was on this trip that I met Charles Moore. He and I talked for half an hour about possibilities. I had a hard time believing that this was the creator of Forth. I played with a Novix development system, unaware that the chip was not yet real, in the sense that one could not buy one. In truth, I felt I was sticking my neck out by suggesting a Forth design when other engineers wanted my head for what I had accomplished in the testing area. The reality was, it did not matter—I ordered a Novix chip which was never delivered, since the company eventually folded. I felt relieved. I went on to work with DSP processors such as the TMS320C25, which were now capable of implementing complex demodulation designs and provided me with new areas to explore.

Then the Berlin Wall fell. The defense buildup was over in a day, but it took several years of excruciating pain for many innocent people to accept the change in their lives. I held out until September of 1991, when I finally admitted it was time for me to leave. I could no longer pay the price required to survive. In January of 1989, I had replaced my aging Victor 9000 with a Dell 386 computer. I briefly went into consulting with my Forth system. I worked several months for the Physical Optics Corporation in Torrance, California, automating their production holographic testers. I realized again that I was sticking them with a custom solution that could not be updated without me. It was just not viable. Even though they were delighted with the results of my work, I never got called back; probably because the engineering staff had changed in the interim.

I was out of work until May of 1992, when I got a call from Systron Donner in Concord, California. A half-dozen

The Forth development environment was so efficient that the H-P BASIC controllers were made obsolete. This led to resentment...

Litton refugees had found jobs there, and they were eager for me to join them. I moved from Los Angeles to beautiful Contra Costa county, and thought I had found a wonderful place to work. The CEO was Dick Terrell, who came from Litton marketing, and was converting the company to quartz sensor technology. It turned out that I was the last one hired before the defense downsizing began in earnest at the company. I had to relive the experience at Litton during the next year and a half.

I was hired to do DSP software for the Quartz Inertial Measurement Unit, but the military requirements for software quality control were beyond the resources of the

company, so the project was canceled a month after I arrived. Instead, I was asked to work on a substitute IMU for a contract delivery that was not going to happen on the current schedule. One of the main problems was that the rate table test station, which was being coded in C, would not be ready in time. I volunteered my Forth system for the interim station, and completed the work in several months. Once again, I experienced the wrath of engineers who said I cheated because they were forced to use the "correct C approach," while I used this thing called Forth, which obviously made the work too easy. Go figure. I should have known better; the truth was, nothing mattered, because the company was being downsized with a vengeance, and when the CEO was replaced, I soon lost my job in December of 1993.

Among the people who joined me going out the door was a guy who wanted to start a company with a product, based on the Systron Donner rotation sensor, which would measure body movements for medical purposes. I met with him and agreed to program a prototype piece of equipment using my Forth system, in exchange for a future piece of the company. In one month, I had a prototype that displayed real-time rotation movement and medical parameters for Parkinson's syndrome. It was demonstrated to the Parkinson's institute and was well received. However, I told my partner that the final device could not be programmed in Forth. Why would I say such a thing? Simply because technology had passed Forth by, in my opinion. It was no longer possible for one person to develop all the software required in a graphical environment. I needed to buy the tools I needed to leverage my time for a professional job. I could no longer play the role of the maverick programmer, nor did I want to. I need to be part of a collaborative community in which I can give and receive work. I do not see Forth as a viable solution as of today.

The startup company never happened, for financial reasons, so I have been unemployed since then. I am also forty-two years old, and am looking at my life from the midway point. I have spent nearly twenty years doing this Forth thing, and I do not know if I want to continue with it. A year ago, I bought the Symantec C++ development package for Windows. I have yet to use it. It does not excite me like Forth did, because it does not give me the freedom to create those program constructs which enable my ideas. I guess I am still undecided on the issue of Forth, so I will renew my subscription to *Forth Dimensions* for at least this one last time.

A Novel Approach to VLSI Design

C.H. Ting, Charles H. Moore
San Mateo, California

OKAD is a simple yet very powerful software package for VLSI chip design. It runs on a 386-class personal computer and requires very few resources. It contains a layout editor to produce and modify IC layout at the transistor level. It can display the layered structures of an ASIC chip with panning, zooming, and layer-selecting facilities. It also includes a simulator which simulates the electrical behavior of the entire chip. Its functionality was fully verified in the production of several high-performance microprocessors and special signal-processing devices.

Introduction

OKAD is a software package that aids silicon layout and simulation. It currently runs on a 386 and adds about ten Kbytes in size. Its purpose is to design full, custom, VLSI chips and produce standard output files suitable for IC production.

The chip design is based upon the actual, geometric layout of five layers. This is distinct from normal practice, where designs are based upon a schematic. OKAD does not use or produce schematics and net lists. Of course, the designer may use them outside the system.

It provides VLSI technology to anybody who wants to transport his imagination to real silicon.

This approach is a matter of personal choice. Silicon compilers, schematic capture, and auto-routing are being explored; other alternatives are not. It is interesting to draw and modify brightly colored graphic images containing a chip layout, and fun to animate them by simulation. The Forth computer language is the foundation of this ambitious software project, and provides the means to achieve the goal of designing an efficient Forth microprocessor.

As is often the case, available tools influence the design. For example, OKAD can properly simulate transmission gates, which encourages their use. Conventional VLSI CAD systems cannot handle transmission gates very well, and designers are discouraged from making use of them. Conventional CAD systems rely heavily on cell libraries which encapsulate designs at the transistor level. The circuits inside the cells cannot be optimized for specific situations. The resulting IC tends to be bulky and inefficient. OKAD encourages the designer to examine each transistor and optimize it for its purpose.

OK, the Graphic Environment

OK is a software interface to a computer. It is derived from Forth and takes its name from Forth's terminal prompt. OK appears on most screens as a key that returns to the previous menu. OK has the capabilities of Forth, but is simpler. It does not use the disk, since computers have large memories. It has no editor or compiler, because it composes and displays object code. It has no interpreter, but is menu-driven from seven keys. It has no multitasking.

OK has been evolving for five years along with OKAD. It is a sourceless programming system that displays code by decompiling. This eliminates the syntactic difficulties that source code encounters—even Forth source. It has run on the Novix 16-bit, ShBoom 32-bit, and 80386 processors, and is destined for my MuP21 processor. With the elimination of source code, a QWERTY keyboard is no longer required. Rather, a seven-key pad or a three-key chording pad is a simple, friendly device. Use it to select among seven menu entries, and you have the good features of a pointing device without the complexity of a mouse.

The 386 version of OK runs under DOS with a VGA display. In a 65 Kbyte segment, about 2K is object code, 8K is tables, the rest is free. There are seven displays of 20 x 15 characters in 16 colors. With them, you define your own words, menus, and screens. Keys are multiplexed by moving through a menu tree. The most common key function is to select another menu. In effect, the space-multiplexing of a large keyboard is replaced by time-multiplexing a small one.

Characters are in 32- x 32-dot matrices. Besides letters, numbers, and some punctuation, 16 graphic symbols are defined for OKAD to compose transistors and IC circuitry. A symbol editor is included, so users can modify the symbols to suit their applications.

Chip Layout

A chip is represented as an array of tiles. For example, the MuP21 microprocessor die is 2.4 mm square. It is formatted as a 600 x 600 array of 4- x 4-micron tiles. Each tile uses four bytes of memory, so the chip uses 1.5 Mbytes.

The present version uses five layers to represent well, diffusion, polysilicon, metal-1, and metal-2. Each layer uses four bits of the tile to choose one of 16 patterns: blank, horizontal, vertical, corner, contact, etc. A tile can form a transistor by itself. It can also be part of a larger transistor. It can also provide electrical connections between transistors and other devices.

A VGA display provides 640 x 480 pixels of 16 colors. It is formatted into a 20 x 15 array of tiles. Each tile may be used to represent a 32 x 32 character or a tile containing patterns in eight colors:

well	gold
diffusion	green
poly	red
metal-1	blue
metal-2	silver

Bright green, red, and blue label nets at 5 volts, as opposed to ground, as determined by the simulator.

The layers are stacked in their physical order. They may be peeled off to examine detail otherwise concealed. Transparent colors are not adequate to look at a design five layers deep.

The designer works with these tiles. The seven keys are programmed to provide a variety of actions:

Pan through image
Move cursor through image
Move cursor through layers
Scroll patterns at cursor
Drag trace through image
Copy, reflect, or rotate region of interest.
Display capacitance of net

With these tools, the designer can construct and connect transistors, compose gates, construct and replicate registers, and finally construct an image of a chip.

With such a layout tool, it is practical to hand craft chips. The advantage of manual place-and-route is that you know what you get. If there is no room for a gate, or if a trace is unfortunately long, you can reconsider the design. The goal is a clear, compact layout and you can continually evaluate your progress. Such an approach is most useful for microprocessor or memory layout, as well as for random logic.

Layout Display

To view the actual geometry, as well as verify the rectangle decomposition, keys are defined to:

Display rectangles
Superimpose various layers
Zoom from full chip to tile scale
Pan around chip

The five design layers are expanded to nine output layers:

Well
n+ active
p+ active
Polysilicon
Metal-1
Contacts
Metal-2
Vias
Passivation

It is very reassuring to view these layers and verify the expansion from the tiled representation of a chip to a geometrically correct layout.

Net Lists

The first step in verifying a layout is to extract the transistors and the nets to which they're connected. The MuP21 is in 1.2-micron CMOS with 6500 transistors connected to 2500 nets. Each transistor is characterized by a drive (uA) and each net by a load (fF).

To facilitate net identification, the program first traces the two largest nets, power and ground. It starts at the input pad and uses a recursive algorithm to follow the trace through metal-1 and diffusion, branching as required. It marks each tile with a flag:

01	power
10	ground
00	neither

It then scans the poly layer and locates transistors where poly crosses diffusion. It measures their size by following the poly trace. It then identifies the nets for source, gate, and drain. It can distinguish source and drain only when source is power or ground. The result is an eight-byte table:

Source net index
Gate net index
Drain net index
Drive

To identify a net that is not power or ground, it follows the trace doing two things:

1. Computing capacitance based on fF/tile for each layer:

n-diffusion	13.6 fF
p-diffusion	12.1
gate	7.4
poly	.7
metal-1	.6
metal-2	.5

2. Looking for a tile bit indicating the 'owner' of a net. If it finds an owner, it searches an eight-byte net table to identify the net:
   Location of owner (three bytes)
   Load

Otherwise, it creates an entry for a new net, with the owner being the location where the search was started. A special case is an internal (series) node in a NAND or NOR gate with capacitance/tile:

p-diffusion 8.0 fF
n-diffusion 8.8

Simulation

Armed with transistor and net tables, the program can simulate the chip. Apply five volts to the power net and observe the consequences. Because all the nets have a capacitive load, there is no DC bias matrix to solve. Simply integrate the differential (difference) equations:

I = u(s,g,d)
dV = I*dt/C

First, calculate the currents into each net from a transistor model. Then adjust the voltage on the net from the current into it, its capacitance, and the time step. Repeat indefinitely.

As with any model, you don't include unnecessary detail. Thus, the poly resistance (80 Ohms/tile) is not included, since it is negligible. Arithmetic is in low-precision integer (16 bit), a version of fuzzy logic.

The transistor model is:

I = K*(2g-d/delta)*d

where

d drain-source (voltage)
g gate-source-body-threshold (voltage)
K bulk parameter
delta 5 for n-transistors, 1 for p-transistors

Originally, voltages are in the units of mV. However, 6400 mV = 4096 units replaces a divide with a shift and requires only two multiplications per transistor. This pragmatic model closely fits measured IV curves from the manufacturer's data. A display exists for manually fitting parameters. The parameters reported in the SPICE programs are much less accurate, and do not produce IV curves measured from silicon.

The time step wants to be large for speed, but is limited by the smallest capacitance. In order to ensure that the voltage change on an internal node is about one volt, it must be 32 ps. It can be variable, since signals mostly change during clock ticks, but that doesn't improve the computation. (Simulation is slow on a 386; the same algorithm running on MuP21 will be ten times faster.)

While a simulation runs, four scope traces can be displayed. Merely point to a metal portion of four nets to select the signals. Rise times, phasing, amplitudes, and glitches are easily determined. Four traces seem to track the simulation adequately.

Having run a simulation, the final signal levels (above or below 2.5 volts) are indicated on the tile display. Now there are 2500 signals sampled at the same time. In particular, you can check the logic and sense of control signals. It allows the designer to exchange NAND and NOR gates freely, and to add to or delete from the number of inverters in a signal chain.

A future enhancement will record the time of transition (through 2.5 volts) for each signal. This will allow easy verification of phasing of control signals relative to the clock. It is also an example of continual improvements you can make if you own the software.

Tape-out

The geometry so far has been purely graphic. The four-micron tiles determine the model for loads and drives. But, basically, the layout is scalable, in that the tiles can be expanded or compressed.

The trace widths for each layer are specified by the design rules. Tile size must be chosen so that separations are adequate. This is inevitably the separation between trace and adjacent contact. With four-micron tiles, this is met except for metal-2, where traces may not be adjacent to vias.

The simplest GDSII (or CIF) tape is composed of rectangles. The tape-out routine scans each layer horizontally (vertically for metal-1) and composes the largest rectangles for each trace, and then writes the rectangles to the tape file. A second scan extracts contacts and vias. In the case of vertical traces, it's necessary to mark visited tiles to avoid revisiting them.

The MuP21 layout generates 65,000 rectangles. The OKAD internal format records two bytes for each of four coordinates (x and y for lower-left and upper- right corners) or eight bytes/rectangle. This is then expanded to 20–30 bytes in the standard GDSII or CIF formats, and then is ZIPed to fit on a floppy to be sent to the foundry.

Conclusion

OKAD is an unconventional VLSI design tool which allows individual designers to design large, custom ICs and to explore ways of optimizing the design. It runs on very inexpensive personal computers, and avails VLSI technology to anybody who wants to transport his imagination to real silicon. It has been used to produce a number of high performance ASIC chips, including MuP21, a 20-bit microprocessor with a peak execution speed of 80 MIPS. It demonstrates that VLSI technology as practiced in the IC industry does leave lots of room for improvements, in spite of the great success in the last 20 years. It also points out a new direction for individual IC designers: that smaller, faster, and better ASIC chips can be designed and perfected without big, complicated, and expensive software tools running on big and expensive mainframe computers or fancy workstations.

Dr. C.H. Ting, a long-time, noteworthy figure in the Forth community, may be reached via e-mail at Chen_Ting@umacmail.apldbio.com or by fax at 415-571-5004.

Charles Moore is the inventor of Forth, an explorer in language and silicon technologies, and the owner of Computer Cowboys.

Forth in Control

Ken Merk
Langley, British Columbia, Canada

The following article is my attempt to contribute back to the Forth community some of my experiences in solving hardware interfacing problems using Forth. My formal training is in hardware, so I consider myself a beginner when it comes to Forth or any software-related projects. I found Forth a powerful tool to interface the computer to the outside world. Its speed and interactive qualities let you build "modules" that can be tried out with immediate feedback from your hardware. Once a solid foundation of primitive words is established, building up from there goes quickly.

The Forth package I use is F-PC by Tom Zimmer, which I originally adopted to learn 8086 assembler language. My first try at an application using F-PC was very successful. Using SMENU.SEQ as a foundation for pull-down menus (with mouse interface), and words like `BOX&FILL`, `SAVESCR`, and `RESTSCR` for pop-up windows, `LINEEDITOR` for text input with full edit capabilities, `FUNKEY.SEQ` for function-key input, and `MCOLOR.SEQ` to make it pretty, it turned out to be a professional-looking program. I thought to myself—just think what I could do if I knew what I was doing!

Using the PC, a Forth disk, and a few electronic components, we can build a simple interface to control devices in the real world. The first step is to get some I/O lines out of the computer. There are four basic routes we can go:

1. Use the expansion slots located on the computer's motherboard. To do this, we need to build an address decoder to select some specific I/O port space, and some latches or PIA chips to get our I/O off the data bus into the outside world. There is 1K of port addresses available here, enough for any situation we can think of. This is the most versatile and complex to implement.
2. We could use the RS–232 port to get control data in and out of the computer. This again involves building some electronics into our interface. We will need a serial-to-parallel converter to change the serial bitstream into useful parallel control data. A UART and baud rate generator would work for this application.
3. The PC game port is another pathway into your computer. We can sense four digital inputs and four analog inputs. There is no provision for data output, so it limits us to just receiving data from the outside world.
4. The parallel printer port is the easiest and least complex to implement. We can bring eight outputs to the outside world for control purposes. The port is also bi-directional, so we can implement some input lines to receive data.

To keep this project as simple as possible, we will use the parallel printer port and build an interface to control eight devices. For now, the eight devices will be LEDs (light-emitting diodes), which will represent the on/off state of each bit on the port.

The first thing we need to do is determine what port address is assigned to your parallel printer card. The three possible parallel ports referred to as LPT1, LPT2, and LPT3 are supported by three base addresses. At boot up, the BIOS searches for parallel ports at each of the three base addresses. The search is always performed in a specific order.

1. Location 03BC (hex) is polled first. A byte is written to address 03BC and then read back to see if it matches what was sent.
2. Location 0378 (hex) is polled second.
3. Location 0278 (hex) is polled last.

The first port that is found is assigned the name LPT1, the next one is assigned LPT2, and the last one is LPT3.

When you first turn on your computer, the BIOS displays an information screen which tells you the addresses of your parallel ports and LPT assignments. If your BIOS does not support this feature, you can use the System Information utility in PC TOOLS, under the I/O Port heading, to obtain this information. When all else fails, we can use Forth to get this information. On boot up, the address of the first parallel port found is stored in address locations 0040:0008 and 0040:0009. To view it, we can use Forth's dump routine.

```
HEX  0040 0008 10 LDUMP
```

A chart will be displayed showing 16 consecutive address locations, starting from location 0040:0008. Address loca-

tion 0040:0008 will contain the least significant byte, and location 0040:0009 will contain the most significant byte. This is the address you will be using to send data to the printer port (LPT1). If the address is 0000, you have no parallel printer card in the system. To check continuity to your port, we can conduct a short test. Let's suppose your port address was shown to be 378 hex. We can write a byte to it and read it back to see if it matches what we wrote. Type the following words:

```
HEX
: WRITE       378 PC! ;
: READ        378 PC@ . ;
```

To write the byte FF to the port, type:

```
FF WRITE
```

To read that byte back, type:

```
READ  -----> FF
```

Try again with different values:

```
00 WRITE
READ  -----> 00
```

The results we get indicate continuity to the printer card.

Building the Interface Hardware

The next step is to build the interface cable and LED readout display. All components needed are available at your local Radio Shack or electronics supply outlet. Here is a list of materials:

1	Solder-type DB25 male connector (276–1547)
30 ft.	#22 gauge stranded hook-up wire (278–1296)
1	Multipurpose breadboard (276–150)
8	470 ohm 1/4 watt resistors (271–1317)
8	Red LEDs
6	Plastic tie wraps

Tools needed:
Pencil-type soldering gun
Rosin-core solder
Wire strippers/cutters

Measure and cut nine pieces of wire, each three feet long. Strip both ends of the wire and tin. Solder a wire to each pin of the DB25 connector, as indicated below.

DB25 Pin#	Bit #
2	0 (least significant bit)
3	1
4	2
5	3
6	4
7	5
8	6
9	7 (most significant bit)
25	Ground

The DB25 connector should have nine wires attached to it on pins 2–9 and 25. Attach tie wraps, equally spaced along the cable, to keep the wires bunched together.

Install and solder the eight LEDs onto your breadboard. Mount them all in a row, with ample space between them so they are not crowded together. The LEDs are polarity sensitive, so they all must be installed in the same direction to function properly. The *cathode* is identified by the flat spot on the rim of the LED. If the LEDs are new and have not been trimmed, the cathode lead will be the longer of the two. All the cathodes will be commoned together and connected to ground (pin 25).

Install and solder a 470 ohm resistor above each LED, as indicated in the drawing *[see pages 23–24]*. One side of each resistor will be connected to the *anode* of the LED below it. The other side of each resistor will be connected to the appropriate wire from the printer port.

Lay the board down in the position in which you are normally going to view it. To stay with convention, the LED on the far *right* of the board will be the least significant bit. Solder the wire from pin #2 of the DB25 connector to the resistor feeding this LED. Continue from right to left, soldering wires pin #3 – #9 to each resistor, the last being the most significant bit. Finally, solder the wire from pin #25 to common cathode bus (ground).

We can test the board before hooking it up to your computer, to ensure that it works properly. To do this, we need a standard 9 Volt battery and a battery clip with power leads. Attach the battery to the battery clip. Clip the black wire (neg.) to pin #25 on the DB25 connector. With the red wire (pos.), touch pins #2 through #9 on the DB25 connector; each corresponding LED from right to left should light up. After testing, remove battery from clip, and disconnect the black wire.

If the board does not function properly, recheck wiring from the DB25 connector to the board. Check the polarity of all LEDs, and make sure all connections look good and that there is no solder bridging the copper traces. If the board looks correct, clean the copper traces with alcohol and a stiff bristle brush to remove dirt and excess flux.

With the board functioning properly, we can connect it to our computer. Plug the DB25 connector into your parallel printer port and turn on the computer. While it is booting up, you will see some of the LEDs turning on. This is normal, as the computer is searching for active printer ports. Run F-PC, and at the "ok" prompt, type `FLOAD FCONTROL.SEQ`. FCONTROL.SEQ automatically searches for an active LPT1 port and assigns the port address to the constant `#PORT`. If no active port is found, the error message "Parallel printer port not found" will be displayed. If no errors are encountered, we can try some control words.

Type `ALL-ON`	All the LEDs should come on.
Type `KILL`	All the LEDs should go off.

In the following section, we will walk through the FCONTROL.SEQ code to see what makes it tick.

Parallel Printer Port Interface

The parallel printer port on the PC has eight outputs that can be brought to the outside world for control purposes. The following Forth code will be used to interface the port to external hardware. Each output line will drive an LED to indicate its status.

Let's pretend your computer is controlling machinery in a factory. We will name each line and assign it a number according to its binary weighting on the port.

```
DECIMAL                          \   Binary
                                 \   weight:
    1    CONSTANT FAN            \   00000001
    2    CONSTANT DRILL          \   00000010
    4    CONSTANT PUMP           \   00000100
    8    CONSTANT SPRINKLER      \   00001000
   16    CONSTANT HEATER         \   00010000
   32    CONSTANT LIGHT          \   00100000
   64    CONSTANT MOTOR          \   01000000
  128    CONSTANT VALVE          \   10000000
```

We will now make some words that will control each bit individually, so we can turn on and off any device we want.

#PORT will be one of the three valid parallel port addresses. b will be the control byte containing device on/off information.

```
CODE  BSET          ( b    #PORT -- )
      POP DX
      POP BX
      IN AX, DX
      OR AL, BX
      OUT DX, AL
      NEXT
END-CODE
```

The word BSET (Bit-Set) will *set* each bit on the parallel port (#PORT) that matches every high bit in byte b. It reads the status of the port and does a logical OR with byte b. The result is written back out to the port. So any bit (device) you want to turn on, you make high in byte b.

```
CODE BRESET         ( b     #PORT -- )
     POP DX
     POP BX
     NOT BX
     IN AX, DX
     AND AL, BX
     OUT DX, AL
     NEXT
END-CODE
```

The word BRESET (Bit-Reset) will *reset* each bit on the parallel port (#PORT) that matches every high bit in byte b. It reads the status of the port and does a logical AND with byte b. The result is written back out to the port. So any bit (device) you want to turn off, you make high in byte b.

```
CODE BTOGGLE        ( b     #PORT -- )
     POP DX
     POP BX
     IN  AX, DX
     XOR AL, BX
     OUT DX, AL
     NEXT
END-CODE
```

The word BTOGGLE (Bit-Toggle) will *toggle* each bit on the parallel port (#PORT) that matches every high bit in byte b. It reads the status of the port and does a logical XOR with byte b. The result is written back out to the port. So any bit (device) you want to toggle, you make high in byte b.

The above code for BSET, BRESET, and BTOGGLE uses the logical functions OR, AND, and XOR as masking templates to preserve the status of the devices we do not want to change. If we send out the byte that corresponds to the device weight to the port, we would activate that device and turn the rest off. By using this masking scheme, we preserve the status of the other devices and activate only the one we want. So, before each command, the port status is read and then masked against the binary weight of the device, and then sent to the port.

```
10110001        current port status
00000100        binary weight for PUMP

--------        To turn on pump (bset), we will do
                a logical OR mask.

10110101        Byte written to port. Pump bit is on
                and the rest have not been changed.
```

Control Word Set

Each device can be controlled simply by commanding it to be on or off:

```
: >ON   ( b --- )          #PORT BSET ;
```

With the word >ON we can activate any device on our port.

```
MOTOR  >ON
```

will turn on the motor

```
FAN    >ON
```

will turn on the fan

```
: >OFF   ( b --- )         #PORT BRESET ;
```

With the word >OFF we can shut off any device on our port.

```
MOTOR  >OFF
```

will turn off the motor

```
FAN    >OFF
```

will turn off the fan

(Text continues on page 22.)

```
\ FCONTROL.SEQ                                  Ken Merk  Apr/95
\ F-PC
\                 Forth Code to control parallel printer port.
\                 ********************************************

DECIMAL

  $0040 $0008 @L               \ Look for active LPT1 port
          0= #IF               \ If no port found then abort

          CLS
          23 8 AT .( Parallel printer port not found.)
          CLOSE QUIT

             #ENDIF

$0040 $0008 @L CONSTANT #PORT  \ Find port addr for printer card
                               \ assign to constant #PORT

1   CONSTANT FAN               \ assign each device its
2   CONSTANT DRILL             \ binary weighting
4   CONSTANT PUMP
8   CONSTANT SPRINKLER
16  CONSTANT HEATER
32  CONSTANT LIGHT
64  CONSTANT MOTOR
128 CONSTANT VALVE

code bset  ( b  #port -- )     \ will SET each bit in #port that
     pop dx                    \ matches every high bit in byte b.
     pop bx
     in  ax, dx
     or al, bx
     out dx, al
     next
end-code


code breset  ( b  #port -- )   \ will RESET each bit in #port that
     pop dx                    \ matches every high bit in byte b.
     pop bx
     not bx
     in  ax, dx
     and al, bx
     out dx, al
     next
end-code


code btoggle   ( b  #port -- )  \ will TOGGLE each bit in #port that
     pop dx                     \ matches every high bit in byte b.
     pop bx
     in  ax, dx
     xor al, bx
     out dx, al
     next
end-code
```

(Code continues on next page.)

```
: >ON       ( b  -- )       #PORT bset    ;      \ turn ON device
: >OFF      ( b  -- )       #PORT breset  ;      \ turn OFF device
: TOGGLE    ( b  -- )       #PORT btoggle ;      \ TOGGLE device

: KILL      ( -- )          00 #PORT pc!  ;      \ turn OFF all devices
: ALL-ON    ( -- )          $FF #PORT pc! ;      \ turn ON all devices

: ON?       ( b -- f )      #PORT pc@ and 0<> ;  \ get ON status of device
: OFF?      ( b -- f )      #PORT pc@ and 0=  ;  \ get OFF status of device

: WRITE     ( b -- )        #PORT pc!   ;        \ WRITE byte to port
: READ      ( -- b )        #PORT pc@ . ;        \ READ  byte at port

: BINARY    ( -- )          2 base !    ;        \ change base to binary
```

```
: TOGGLE   ( b --- )     #PORT BTOGGLE ;
```
The word TOGGLE can be used to change the status of any device on the port. If the device is on, a TOGGLE command will turn it off. If the device is off, a TOGGLE command will turn it on. This command can be useful to create digital pulses. It will take the present condition of port bit and invert it for a selected time, then toggle it again to the original condition.

```
FAN TOGGLE
```
If fan is on, the command will turn it off.
If fan is off, the command will turn it on.

The words ALL-ON and KILL will control the states of *all* devices:

```
: KILL   ( --- )       00 #PORT PC! ;
: ALL-ON   ( --- )     $FF #PORT PC! ;
```

The word KILL can be used to shut down all devices on the port. It can also be used at the beginning of the program to clear the port to a known condition.

```
ALL-ON
```
will turn on all devices

```
KILL
```
will turn off all devices.

We can now make some words that will check the status of each device. After a status check of a device, a branch can occur depending on the flag value.

```
: ON?    ( b --- f )   #PORT PC@ AND 0<> ;
: OFF?   ( b --- f )   #PORT PC@ AND 0=  ;
```

```
FAN ON?
```
Will return a *true* flag if the device is on, or a *false* flag if it is off.

```
FAN OFF?
```
Will return a *true* flag if the device is off, or a *false* flag if it is on.

To turn on or off any combination of devices, we can use the following code:

```
: BINARY   ( --- )       2 BASE ! ;
BINARY
11110000 WRITE
```

The output port will match the binary byte. A "1" will cause the LED to be on. A "0" will cause the LED to be off.

```
READ
```
will show the status of the port (e.g., 11110000).
Note: Any error will cause BASE to go back to DECIMAL.

Here are some other commands we can try. On the same line, type:

```
MOTOR TOGGLE  2 SECONDS  MANY
```
The MOTOR LED will blink on and off every two seconds. To end the cycle, hit any key.

```
MOTOR >ON  2 SECONDS  MOTOR >OFF
2 SECONDS  10 TIMES
```
The MOTOR LED will come on for two seconds and then off for two seconds. This will be repeated ten times.

To speed it up, type:

```
MOTOR >ON  100 MS  MOTOR >OFF
100 MS  10 TIMES.
```
The on/off time has been changed to 100 milliseconds.

To build more complex control structures, we can incorporate multiple devices in the control byte. In our imaginary factory, we have a large mixing tank that needs to be cleaned out at the end of the day. To do this, we open up the VALVE at the bottom of the tank and PUMP water into it. After the tank is flushed out, we turn off the PUMP and

close the VALVE. The control sequence could go like this:

```
VALVE >ON   PUMP >ON   15 MINUTES
PUMP >OFF   VALVE >OFF
```

We can make a control word called FLUSH that will both turn on the pump and open the valve:

```
132 CONSTANT FLUSH          \ 10000100
```

The new control sequence could now go like this:

```
FLUSH >ON   15 MINUTES   FLUSH >OFF
```

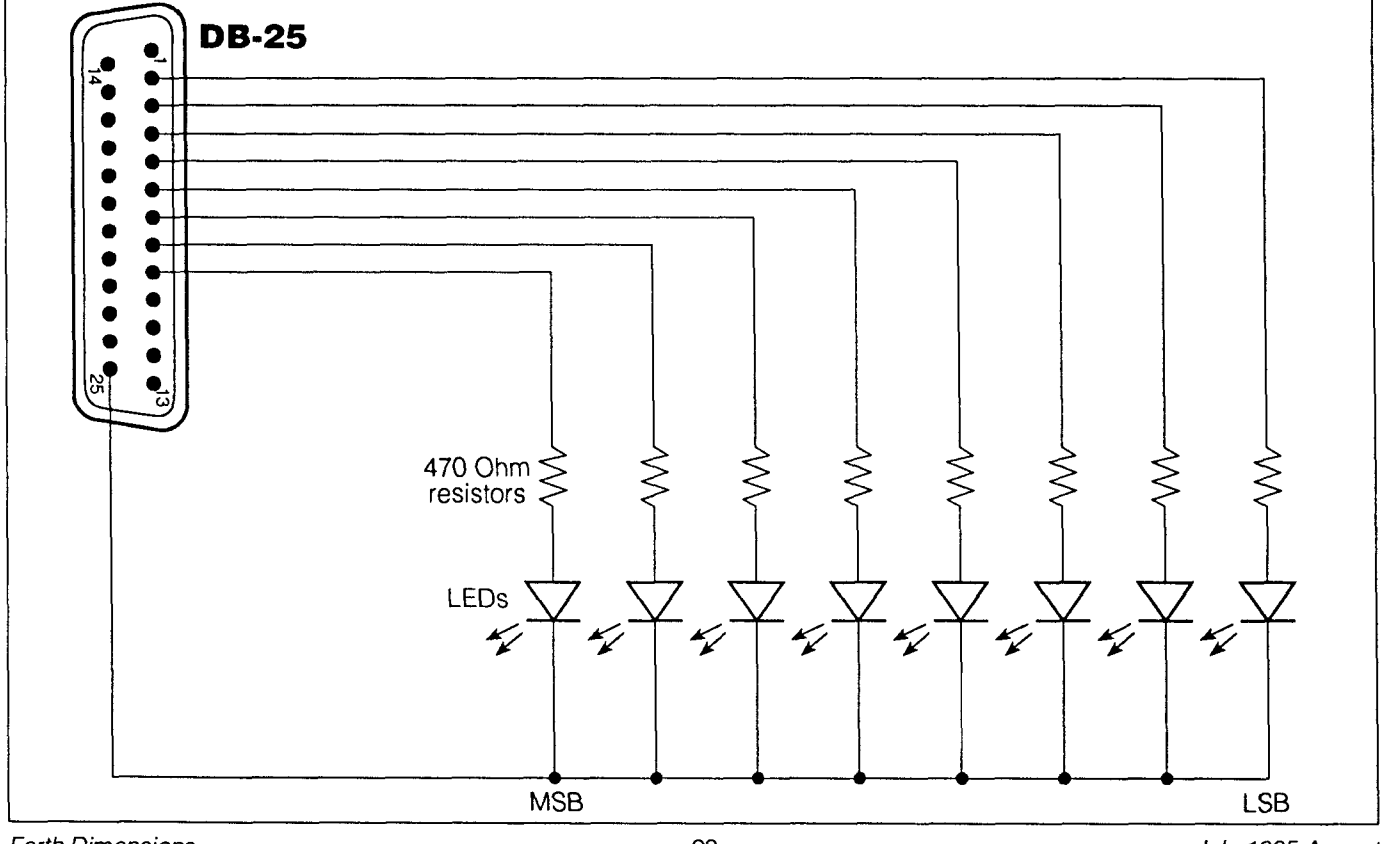
Using this simple concept, we can build very complex control structures that are very "readable," so a non-technical person can understand a sequence of control commands and even write them.

Make up your own commands and try them out. Maybe next time, we can hook up a stepper motor to our interface or some infrared diodes for remote control applications.

P.S. A special thanks to Tom Zimmer who gave us that huge pile of F-PC code to play with, for all that time he spent staring at his computer monitor.

Ken Merk, who graduated from BCIT as an Electronic Technologist, is a married father of two girls and lives in Langley, B.C., Canada. He works for Canadian Pacific Railway, and is involved in a braking system used on caboose-less trains—the caboose is replaced by a black box which monitors many parameters of the train and sends them digitally by radio to the head end. In emergencies, a remote radio can trigger braking. Other projects include infrared bearing-failure detectors, wind detectors, and mountain-top radio communication sites. Merk originally used Forth to learn 8088 assembler, and found it a great tool to control electronic hardware.

Schematic. (Also see next page.)



Parallel-port interface.

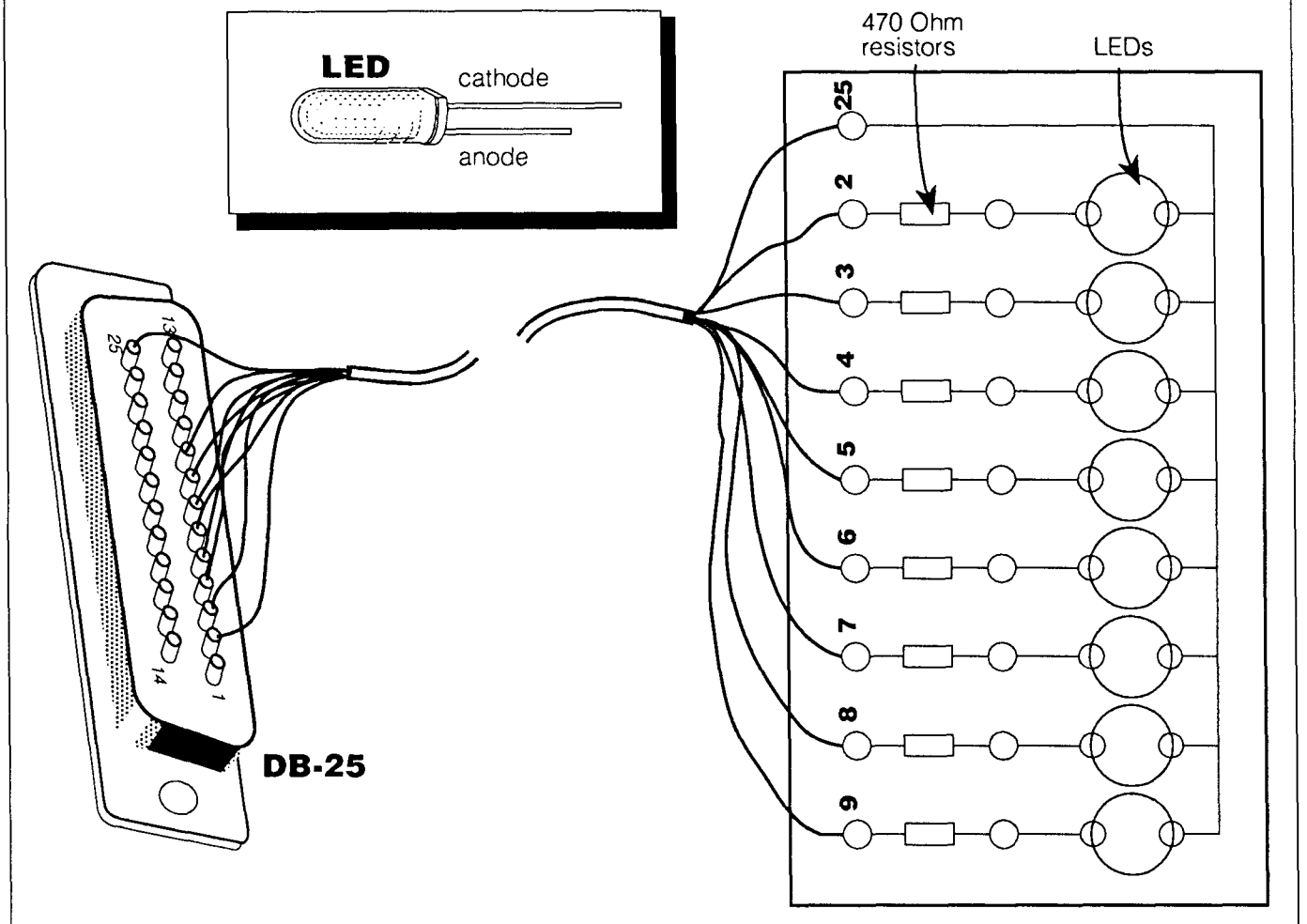


FIG Board Elections

The Forth Interest Group recently held elections for its Board of Directors. The Board was previously composed of seven members, but recently moved to increase in size by two seats; all nine of the positions were up for election. Ten people had accepted nomination to run for a Board position, thereby necessitating an open election. All active members of FIG were eligible to vote—ballots and candidates' statements were mailed with the last issue of *Forth Dimensions*.

The results were counted on June 5, 1995 by members of the Silicon Valley FIG Chapter, and the count was verified by FIG Secretary Mike Elola. The election results, listed in order of the percentage of total votes received, are as follows:

1) Everett "Skip" Carter (91%)
2/3) Elizabeth Rather (88%)
2/3) Brad J. Rodriquez (88%)
4) Jeff Fox (84%)
5) Andrew McKewan (81%)
6) John D. Hall (80%)
7) Mike Elola (78%)
8) Al Mitchell (74%)
9) Nicholas Solntseff (61%)
10) Jack Woehr (53%)

Our congratulations go to each member of the new FIG Board, along with our thanks for the dedicated service of outgoing directors Dave Petty, Dennis Ruffer (Treasurer), C.H. Ting, and Jack Woehr (Vice-President).

New officers had not been appointed at press time, but any changes in those positions will be reported in *FD* at the earliest opportunity.

Those wishing to contact the Board may address their correspondence to:

Forth Interest Group
att'n: Board of Directors
P.O. Box 2154
Oakland, California 94621
Fax: 510-535-1295

Code Size, Abstraction, & Factoring

John Wavrik
San Diego, California

Editor's note: This material was originally part of a discussion about "Code Bloat" that was found on the USENET newsgroup comp.lang.forth.

On comp.lang.forth, Darin Johnson wrote:

> To me, not having been indoctrinated, the difference is that "factor" seems to imply "make things smaller" without anything else attached. "Abstraction" implies "make things better and more usable."

Part of the problem might be a confusion between factoring as a device to make code smaller, and factoring as a device to make code more comprehensible. I suspect it is usually used for the latter purpose.

Suppose a piece of code has a segment which carries out a task which is logically independent of the body of the code. For example, suppose that code for sorting an array of numbers is embedded in the definition of a word. The client word needs to know that sorting is done, but does not have to know *how* it is done—so the sorting routine is factored out. Factoring achieves the effect of separating what is done from how it is done: it is an isolation process. Here it has no impact on code size.

If, later in the application, another sort occurs, the factored-out word could be used, rather than repeating the code. In this case, the factoring does result in a *reduction of code size.*

Suppose that, later in the application, there is also a sort of an array of strings. The programmer then might try to create a general sort routine that will handle both strings and numbers, being passed the address of the array and the address of the comparison routine. This would be *abstraction.* At the same time, it would factor the sort routines out of the places they were originally used, so it would also be an example of factoring. Whether the abstraction would result in smaller code size or in more comprehensible code depends on what is involved in doing it.

Factoring can also play a role in *data abstraction:* it can be used to separate words which need to know how data structures are actually implemented from those which don't. It can increase flexibility by making the bulk of code independent of the choice of data representation. The effect on code size is neutral. It can have the effect of simplifying coding.

In mathematics, there are often analogies between the properties of objects that one would like to exploit in programming (say the similarity of integer arithmetic and the arithmetic of polynomials in one variable over a field). The process of abstraction would be one of generalizing procedures to make them applicable to a variety of objects. Differences in data representation often make it difficult to write simple code to handle abstraction of this sort. (In some cases, one just winds up with several separate processes combined by conditionals.) This type of abstraction does not seem to be related to "factoring." It can often result in making the task of writing the code harder.

> You don't "break" many functions into few, you break few ones into smaller parts. Thus, if you've got a window system with, say, 50 function calls total, and assuming it is nicely abstracted, this is simpler to understand than 100 or 200 functions chosen merely because they were smaller.
>
> Depending upon what you're working with, improving abstraction can either increase or decrease the number of functions. Factoring without abstraction only increases the number. Abstraction may greatly increase the size of a program; for instance, supporting any combination and arrangement of widgets will usually take more underlying code to support than allowing only a few fixed choices.
>
> *—Darin Johnson*

I suppose we all program applications in a hierarchical or stratified fashion. The user may only need to know one word at the top level ("GO"), and can remain completely ignorant of how many words were required to define it.

Suppose level three of an application exports 50 words to level four. If, in writing these 50 words, the programmer uses a lot of factorization, then perhaps 100 or 200 words will be produced in the process of writing level three—but still only 50 words need to be exported to level four. Increasing or decreasing the total number of functions really is not so much the issue as the number of exportable functions (functions used in subsequent programming).

Whether abstraction results in smaller code size or in more comprehensible code depends on what is involved in doing it.

Factoring could increase the number of independently useful words, resulting in both a decrease in code size and a simplification in subsequent programming. On the other hand, some of the factoring might just improve the quality of code at the given level—without any impact at all on subsequent code.

John Wavrik is an Associate Professor of Mathematics at the University of California - San Diego. His research is in the area of Abstract Algebra. He started using Forth in 1980 in an effort to exploit the potential of the microcomputer as a research tool. His interest in computational aspects of Algebra has continued, and he uses Forth to develop research systems in this area. He has also taught a course, using Forth, to introduce pure mathematicians to computers. He can be reached at jjwavrik@ucsd.edu via e-mail.

From FIG Chapters...

Southern Wisconsin FIG Meeting

Reported by Bob Loewenstein
rfl@oddjob.uchicago.edu

The May meeting of the Southern Wisconsin FIG chapter meeting was held at Yerkes Observatory in Williams Bay.

Scott Woods brought the prototype hardware for his ADS/TDS metal detector, and gave a demo of its interface as well as a close look at the hardware. Scott also showed his F-PC local information monitor that he has broadcast on all of his home television sets.

This meeting was Ron Kneusel's first. Bob had corresponded via e-mail with Ron about a project he had done in Yerk, so the meeting provided the opportunity for them to finally meet. Ron showed his 6502 QForth running on a Mac. To do this, he wrote a virtual 6502 machine on the Mac using Mops. Ron also talked about his Forth background, as well as his familiarity with other systems, both hardware and software. He maintains an FTP site at 141.106.68.98 and may have volunteered to create a SWFIG WWW home page.

Bob Loewenstein talked about visiting Mike Hore in Sydney, Australia, in April of this year. He also mentioned his proposed plans to upgrade Yerk to PowerPC native code. He still has not decided whether to use C or assembly to create the kernel. C would obviously be more portable to other platforms, if appropriate hooks were made to isolate Mac-specific code.

Olaf Meding arrived 20 minutes late, due to Olaf's decision to give his front license plate to his father who was visiting from Germany... the cop didn't see it Olaf's way, explaining that front license plates *are* necessary in Wisconsin.

We discussed how we might entice new members to the group. One idea was to try to hold regularly scheduled meetings, as in the past. For the past six months or so, meetings were either not held, or called at a few days' notice (not conducive for big crowds to attend).

At the Madison Expo held in late April, about 40 people signed interest sheets. A follow-up questionnaire was drafted to send to these people. We hope some of them will come to the next meeting, to be held in Madison.

We also talked about a program to introduce Forth to any newcomers at the meeting. We discussed possible applications where Forth is more applicable than other languages, and demonstrating how the program would work in Forth for the next meeting.

The meeting ended around 10:30 p.m. and was followed by a tour of Yerkes, home of the world's largest refracting telescope.

A draft of the questionnaire to interested people is below:

SWFIG Questionnaire

Dear Forth enthusiast(?),
Are you interested in attending the Southern Wisconsin Forth Interest Group (SWFIG) meeting in Madison in June '95?
What is your level of expertise in Forth?
What computers and/or systems are you familiar with?
What programming languages, if any, have you programmed in or are familiar with?
What is your interest in Forth?
Help us prepare for the meeting — what would you like us to discuss?

Silicon Valley FIG Meeting

Condensed from a report by Jeff Fox
jfox@netcom.com

May 27, 1995—Charles Moore said his F21 chip was suffering from too many improvements. The design required some chip-wide changes based on the results of the last P8 prototype run. He said the simulator currently says 400 mips internally on F21 (at one point he had it tuned to 500, but worried he had pushed it too far). Memory access will still limit F21 to less than 300 mips maximum performance in memory.

Chuck said it was about a one-day job to stretch the design of the F21 CPU to make the 32-bit P32 design. He said the P32 would use six 5-bit instructions. One of the two leftover bits would be return, and he didn't know about the other yet.

Chuck said he was using new names for the signals previously called I/O and SRAM on MuP21, that he was now calling these RAM and ROM. Chuck talked about the analog I/O coprocessor, the network interface processor, and the configuration registers on F21.

Chuck also said he had been thinking about nanotechnology. He talked about the statistical distribution of dopants in transistors, and the problems when there are so few atoms in a transistor.

He has been speaking with NASA and the U.S. Air Force

about projects like satellites and the Mars rover. Chuck said he perceives that antagonism to Forth has faded, that they aren't locked into ADA.

Dr. C.H. Ting then demonstrated a multi-voice music program that was using the video output processor on MuP21 to generate the analog signal being played. He showed details of the design and code, and explained that he used a much slower xtal than the one used to generate the video timing for the P21 coprocessor.

Tom Zimmer and Andrew McKewan are about to disappear to the jungle of Texas. This was their last chance to talk about their 32-bit Forth (F95) for Windows to this group for a while.

Andrew wrote the original C wrapper and the kernel of F95. He also ported the object-oriented code from Neon, which he said was useful as an interface to the windowing system. He discussed the issues in hosting an interactive Forth under the Windows environment.

Andrew said he considered the system a good environment to learn about the Windows interface, because you could test anything interactively. But he said he does not really consider it a Windows GUI development environment, because it is lacking too many things. There is no icon editor, no help editor, etc.

In addition to names for Windows functions, and constants for control locations in Windows, F95 contains most of the features of F-PC. The system boots from FORTH.EXE and loads FORTH.IMG. The IMG contains the dictionary and can be from 300K to 2M. The EXE file contains the C wrapper, and changing it would be similar to metacompiling a new kernel in F-PC.

You can simply say

```
' MAIN IS BOOT     FSAVE FOO
```

to save a FOO.EXE and FOO.IMG. You can then install and use it in Windows.

Tom Zimmer extended Andrew's F95 kernel to include all the utilities and tools he needed to finish porting a multi-megabyte Forth project from a fragmented 16-bit Forth system to the 32-bit Windows system. The extensions include assembler, disassembler, debugger, decompiler, editor, class, objects, mouse interface, and graphic interface. A floating-point package has been added, but it requires floating-point hardware.

Tom talked about the OO extension and how, when he only needed one particular new object, he felt it was wasteful to create a new class for it. So he created a way to define such objects without having to first define a one-of-a-kind class. He also talked about headerless classes, and showed how methods are assigned to classes.

Tom explained that the interpreter was a little unusual. First it looks up a word to see if it is in the dictionary, then it tries to convert it to a number, and finally, if it can't do either of these things, it performs a hash and leaves the value on the stack. This is because it assumes that it is a method and that its value should be put on the stack so the object can resolve it at runtime. Objects match the method value to those in their own list of methods to find executable code.

ADVERTISERS INDEX

ANS Forth Clarification Procedures

Greg Bailey (greg@minerva.com)
Hillsboro, Oregon

Editor's note—Here's some help with acronyms for ANSI neophytes: TC = technical committee; X3J14 = the designation of the specific TC that developed ANS Forth; AD = alleged defect(s); X3 = the ANSI body that oversees the activities of various TCs.

The X3J14 TC has set up procedures for handling Requests for Clarification/Interpretation or Alleged Defects in the document X3.215-1994 (ANS Forth).

The most formal procedure is to submit a written request to the X3 Secretariat (X3J14 c/o X3 Secretariat, 1250 Eye St. NW, Suite 200, Washington, DC, 20005–3922; 202-628-2829 or fax 202-638-4922). In this case, the TC's current procedures are as follows:

1. AD will be mailed to all members.
2. Chair will appoint a member to draft the TC's response.
3. Chair will announce the AD and the appointment to all members via U.S. and electronic mail using the mailgroup X3J14@minerva.com.
4. TC members will collaborate with the appointee in composing the draft using appropriate means including primarily the above mailgroup.
5. When Chair is satisfied with the draft, it will be submitted to the TC for letter ballot.
6. Chair will modify these procedures when and as she determines that such is necessary.

When this is done, X3 monitors the proceedings and the TC has specific obligations. Recently, TC chair ruled that we should treat electronic requests similarly, but with streamlined electronic processing during response study and composition.

If you wish to submit a request for clarification, a request for interpretation, or a "bug report" for the Standard, please format your query suitably for publication and e-mail it, *clearly stating at the beginning that you wish the TC to subject it to due process,* to any of the following:

x3j14@minerva.com	(TC only)
greg@minerva.com	(log and distribute to TC only)
ansforth@minerva.com	(widest distribution)

(As long as it is practical to do so, and as long as the traffic is welcome, the ansforth list will subscribe a gateway to comp.lang.forth so that all postings to ansforth will appear on the newsgroup, but not vice versa. Therefore, for anyone who regularly reads the c.l.f, there is no need to subscribe to the mailgroup; just make postings that are suitable business for the mailgroup via e-mail to ansforth@minerva.com, and read them and any public responses on netnews. The advantage of this scheme, again, is that it includes a broader audience in the process than just the usenet news by itself. The disadvantage is that a netnews reader will need to look at the header for any ANS Forth traffic. *Suggestion:* When starting any ANS Forth thread, begin the subject with ANS to warn Usenet readers of need to reply via e-mail [in order for their replies to reach all subscribers to the ansforth list]. In addition, all postings to the ANS Forth list are crossposted automatically to the FIGI-L list.)

When your query has been recognized for processing, you will receive an acknowledgment of that fact. When the TC has completed processing of the query, the reply will be posted to the full ansforth list. The process takes an absolute minimum of thirty days, due to the mail balloting requirement; but having been thus processed, they may be regarded as authoritative. (All inquiries sent to the TC via the ANSForth mailgroup which do not clearly indicate the desire for formal reply will be answered informally by individual TC members and other interested parties.)

Queries and replies will be posted on
ftp://ftp.uu.net/vendor/minerva/x3j14/queries
This server is mirrored in the U.K. and various other places. There is also a Web page at
ftp://ftp.uu.net/vendor/minerva/uathena.htm
which serves to bind the postings together and which includes links to other prominent Forth resources.

Clear identification of quasi-formal inquiries will save all of us time and bandwidth. Thank you very much!

Additional Resources

The Unreal Thing—For those with a legitimate need, the final draft of the Standard is posted as a working document on the root FTP server, ftp.uu.net. Before retrieving and using any of the draft files, please read and comply with the instructions and restrictions defined in the files 00README.TXT and DPANS94.TXT.

The URL ftp://ftp.uu.net/vendor/minerva/uathena.htm provides access to all of the information.

The Real Thing—To obtain the official standard (Document X3.215–1994), please contact:

American National Standards Institute, Sales Dept.
212–642–4900 or fax 212–302–1286
or Global Engineering Documents
800–854–7179 or fax 303–843–9880

Test Suites—John Hayes at Johns Hopkins has written an unofficial but very good test suite for most of the Core wordset. This test suite is posted on ftp.uu.net as above.

It is informal, in the sense that it is not a product of X3J14 nor is it "blessed" by the TC. However, it is very useful. If John releases new versions of his suite, they will be posted in the above archive. Anyone with bugs or improvements for John's test suite is encouraged to e-mail them to ansforth@minerva.com, which will get them to John.

Others with test suites that have the ring of authority are welcome to upload them to ftp.minerva.com:\incoming (anonymous/e-mail) and discuss their posting with greg@minerva.com. Only send material that may be freely distributed legally, with any necessary boilerplate embedded in the source; we can zip and authenticate here, if you wish.

Stretching Forth

Pinhole Optimization

Wil Baden
Costa Mesa, California

The performance of compiled programs can be greatly improved by a little bit of optimization. One standard technique is "peephole optimization." In peephole optimization, the compiler's output is examined for sequences of operations that can be replaced by more efficient sequences.

This is the approach taken in Tom Almy's ForthCMP, Xan Gregg's "PowerMacForth Optimizer" (*FD* XVI/6), Charles Curley's "Optimization Considerations" (*FD* XIV/5), and David M. Sanders' "Optimizing '386 Assembly Code" (*FD* XV/6).

In those systems, the implementation is subroutine threaded or compiled machine language. The implementations are all specific to particular hardware.

Peephole optimization for a direct token-threaded or indirect token-threaded implementation can be provided by having the text interpreter remember the words most recently encountered when compiling. Only the execution token or compilation token of words needs to be remembered. The compiler knows what it did with them.

For practicality, only the last three operations will be remembered.

Just the two previous execution tokens have to be remembered.

This can be done by extending the standard word COMPILE, or equivalent in your system.

```
VARIABLE last
VARIABLE penult
: COMPILE,          ( execution-token -- )
      last @  penult  !
      DUP  last  !
      ( Old Definition of COMPILE, )
COMPILE,
;
```

I call this narrow-window approach to optimization *pinhole optimization.*

In traditional implementations of Forth, an immediate word has at most one execution token or compilation token associated with it. That is, IF may have 0BRANCH, ." may have (."), LITERAL may have do-LITERAL. These tokens may have different names in your system, are often headerless, not found in a wordlist, and known only to the compiler.

With pinhole optimization, such words may have more than one execution or compilation token. The appropriate one will be selected by the optimization logic. Previously compiled tokens may be replaced.

Pinhole optimization extends the number of immediate words. Words that will benefit by being combined with words that have just been compiled are made smart. Execution tokens are provided for *secret words* written in low-level Forth.

As an example of one possibility, some systems have -ROT equivalent to ROT ROT. With pinhole optimization, -ROT may not have a name in any dictionary, but when ROT is encountered by the text interpreter in compilation state, it looks to see if the previous word was ROT; if so, it replaces the compilation token of ROT with the compilation token of the secret word -ROT.

Other examples are not so straightforward, and replacement words usually don't have a meaningful name.

Pinhole optimization is most useful with literals and logic.

When the text interpreter encounters a literal in a definition, it puts out do-LITERAL, or whatever your system calls it, followed by the binary value of the literal. When the definition is executed, do-LITERAL will take the binary value following it, and push it onto the stack.

With pinhole optimization, if the literal is followed by +, do-LITERAL will be replaced with do-LITERAL-PLUS. When the definition is executed, do-LITERAL-PLUS will take the binary value following it, and add it to the top element of the stack.

This roughly halves the time it takes to add a literal.

In a traditional implementation, 0= IF or 0= UNTIL will generate two tokens, that might be called do-ZERO-EQUAL do-BRANCH-IF-ZERO, followed by a destination. FALSE = IF or FALSE = UNTIL will generate three tokens followed by a destination. With pinhole optimization, they will all generate one token, do-BRANCH-UNLESS-ZERO, followed by a destination.

Pinhole optimization would also look for DUP and

other words preceding IF, UNTIL, and WHILE.

This roughly halves, and sometimes quarters, the time to make a test.

My experience has shown that pinhole optimization for threaded implementations generally improves speed of execution about 25 percent. For some applications, particularly those with many variables, the improvement can be as much as 80 percent. Macros, such as discussed is the last issue, can improve 25 percent on top of that.

In the listings here and in the previous "Stretching Forth" articles, certain phrases are underlined. These phrases show where pinhole optimization would occur in a definition.

The following compiles to five primitive Forth instructions.

```
DUP  10 <  NOT IF  10 - [CHAR] A +
[CHAR] 0 - THEN  [CHAR] 0 +

DUP  10 <  NOT IF  7 +
THEN  48 +
```

The following loop is two primitive Forth instructions.

```
BEGIN 1 - ?DUP 0 = UNTIL
```

With pinhole optimization, there is seldom a need for [mumble] LITERAL to optimize an calculation.

One complaint about Forth that I have heard from some Forth programmers is that it doesn't have logical AND and OR.

Here is how they were defined in the last "Stretching Forth":

```
: ANDIF S" DUP IF DROP" EVALUATE ;
IMMEDIATE
: ORIF  S" ?DUP 0= IF" EVALUATE ;
IMMEDIATE
```

In `oliver ANDIF hardy THEN`, optimization makes *two* words out of ANDIF. If `oliver` is false, then `hardy` is not performed.

In `stanley ORIF laurel THEN`, optimization makes *one* word out of ORIF. If `stanley` is true, then `laurel` is not performed.

If POSTPONE were used to define ANDIF and ORIF, optimization would not take place.

Pinhole optimization increases the number of primitive, low-level Forth definitions in an application to improve performance. But the source code does not change. This means you can make some optimizations in an application after you have got it working, without changing the source code.

An optimizing compiler is never complete, as you will be forever thinking of new optimizations to recognize.

Code begins on next page...

WIL BADEN is a professional programmer with an interest in Forth. He can be reached at his e-mail address, wilbaden@netcom.com.

("Backspace" continued from page 39.)

function without affecting the code where it is called. In this case, Forth has the clear advantage.

As for being back to square one when you find a word that the all-so-powerful-kernel writer didn't make deferred, that just isn't true. At the least, all you have to do is make the word deferred, rename the old definition, and set it up to be the default vector. This is exactly analogous to the linking behavior seen in C. In fact, you might even want to argue that it would be a better solution to allow every word to be re-vectorable from within Forth, by default. This would give you even more flexibility than you'd get with C.

Here is why: I think your example in this case is flawed. Often, what you want is not to revector some kernel word, but rather some word used by a kernel word. In this case, you are as out of luck with C as you are with Forth, since the internal functions used by the word you want to alter are not likely to be visible for a relinking fixup. In fact, this is the very strength of component or object-oriented approaches—the internal code is hidden from the user of the interface. You are just as out of luck if the hash-table-in-C object doesn't have a method visible for you to override. It's a completely analogous problem. I think you are being a bit blinded by the power of relinking, and not seeing through to the fact that the problems remain and don't really get any easier to solve.

Take your example of fopen. In many, many C systems, in order to get reasonable performance through the I/O library, functionality that looks like a function call is really a macro. This means that you cannot relink, because you don't really have a function call to revector in the first place. Let's take this one step further with an example. Perhaps you want to add multitasking to your (Forth) system. This involves synchronizing access to global state, such as I/O tables, buffers, etc. Sure, you could provide another layer of functions between your system and the library, so that your intermediate layer can take locks and do all the mutual exclusion stuff it wants to. This is a very big job, but it can be partially automated; it is painful, but doable. But if you want to link with third-party libraries, you are out of luck because they were compiled against the non-thread-safe C-compiler-vendors library and, lucky you, half of the I/O "functions" are really macros. You don't have any way to wedge your replacement functions "in between" the third party code and the C library, because the macro implementations have completely elided the function calls. (This is a real life example from my recent working past.)

In conclusion: *I heartily agree* that a mutable, portable, code base for Forth systems is needed. *I strongly disagree* that C is the right answer to this. As noted above, you are trading away non-portable assembly language for the portability of C, but you are not necessarily doing anything but pushing problems into third-party software, where they are even harder to fix! I believe that a look into a metacompilation-like solution would be very worthwhile, not only because it would allow system generation in the very same language (the duality of assembly and Forth is no better or worse than the duality of Forth and C, *except* there is more inappropriate matching of concepts between C and Forth—C 'for' loops versus Forth's DO WHILE—that can cause cognitive dissonance), but also because you too easily brushed it aside. It may not end up being feasible, but that is neither obvious to me, nor argued persuasively in your column(s). (Granted, I may have missed something more than six months old.)

—Doug Philips (dwp@transarc.com)

Pinhole optimization.

These are samples of the pinhole optimizations in the system I'm using at the time of writing. They are subject to change without notice.

A lowercase letter other than "b" represents a literal, constant, or variable. "b" represents a literal or constant that is a power of 2.

```
n +       x n +     x + n +     0 +     SWAP +
n -       x n -     x + n -     0 -     SWAP -

x n *     b *       1 *         CHARS
 x n /     1 /
 x @     DUP @
 x !     SWAP !
 x +!    SWAP +!
 n <     SWAP <        0 <
 n =     n OVER =      DUP n =     0 =     FALSE =
 n ALIGNED
 n AND         x n AND
 n >BODY       ['] x >BODY           ['] >BODY n CELLS +
 ['] >BODY n CELLS + @               ['] >BODY n CELLS + !
 n CELLS
 n COUNT           C" ccc" COUNT
 DUP IF            ?DUP IF         DUP 0= IF       ?DUP 0= IF
 0< IF             FALSE IF        0< NOT IF       0= NOT IF
 1 +LOOP           1 CHARS +LOOP
 x n LSHIFT        x LSHIFT n LSHIFT     n LSHIFT
 b MOD
 n OR          x n OR
 0 PICK        OVER 1 PICK     1 PICK     n PICK
 ROT ROT       ROT ROT ROT
 n RSHIFT      x n RSHIFT
 n U<          SWAP U<
 n UNDER+
 DUP UNTIL     ?DUP UNTIL      DUP 0= UNTIL      ?DUP 0= UNTIL
 0< UNTIL      FALSE UNTIL     0< NOT UNTIL      0= NOT UNTIL
 n XOR         x n XOR
 DUP WHILE     ?DUP WHILE      DUP 0= WHILE      ?DUP 0= WHILE
 0< WHILE      FALSE WHILE     0< NOT WHILE      0= NOT WHILE
 AGAIN ;       QUIT ;          ABORT ;           BYE ;
```

Here is a typical implementation of the required control-flow words other than LEAVE. Words that contain one or more lowercase letters, or are a single uppercase letter other than I or J, are not in Standard Forth, and probably won't be available to you.

Not all optimizations are given.

branch, compiles part of an unconditional branch;
?branch, compiles part of a conditional branch;
do,, loop,, and +loop, compile parts of the execution semantics of DO, LOOP, and +LOOP.

>mark puts the origin of a forward branch on the control-flow stack;
<mark puts the destination of a backward branch on the control-flow stack;
>resolve resolves the destination of a forward branch;
<resolve resolves the destination of a backward branch.

rake gathers the LEAVEs within a DO loop.

```
: swop      1 CS-ROLL ;

: IF        ?branch, >mark ; IMMEDIATE
: THEN      >resolve ; IMMEDIATE
: ELSE      branch, >mark swop >resolve ; IMMEDIATE
: BEGIN     <mark ; IMMEDIATE
: UNTIL     ?branch, <resolve ; IMMEDIATE
: WHILE     ?branch, >mark swop ; IMMEDIATE
: REPEAT    branch, <resolve >resolve ; IMMEDIATE
: DO        do, <mark ; IMMEDIATE
: LOOP      loop, <resolve rake ; IMMEDIATE
: +LOOP     +loop, <resolve rake ; IMMEDIATE
```

The implementation in your system may not look like that, but you should be able to make a correspondence.

In traditional Forth implementations, `branch,` and `?branch,` look something like:

```
: branch,   do-BRANCH  COMPILE, ;
: ?branch,  do-BRANCH-IF-ZERO  COMPILE, ;
```

We want to optimize these so that there is no penalty when a conditional branch is preceded by `0=`.

We are using *smart* `@`, `!`, `+`, and `,`. They know when they are accessing data space and when they are accessing code space.

`HERE` is the next available address in dataspace.
`next` is the next available location in codespace.

`ALLOT` allocates in dataspace.
`gap` allocates in codespace.

`codes` is like `CELLS` but for codespace.

Words beginning with `do-` are execution tokens implemented in low-level Forth. You'll have to roll your own.

```
( Optimize:  0= IF    0= WHILE    0= UNTIL   )
```

The definition of `branch,` is the same, but uses the new definition of `COMPILE,`.

However `?branch,` becomes:

```
: ?branch,  last @  do-ZERO-EQUAL =  IF   \ 0= IF|UNTIL|WHILE
                -1 codes  gap
                do-BRANCH-UNLESS-ZERO  COMPILE,
            ELSE                        \ IF|UNTIL|WHILE
                do-BRANCH-IF-ZERO  COMPILE,
            THEN
;

( do-ZERO-EQUAL is the execution-token of 0=. )
```

Let's make some more optimizations in `?branch,`.

```
(
Optimize:
      ?DUP 0=  IF     ?DUP 0=  WHILE     ?DUP 0=  UNTIL
      0=  IF          0=  WHILE          0=  UNTIL
      0=  0=  IF      0=  0=  WHILE 0=   0=  UNTIL
      DUP  IF         DUP  WHILE         DUP  UNTIL
```

```
      0  IF          0  WHILE          0  UNTIL
)

: ?branch,
      ( CASE )
            last @  do-ZERO-EQUAL =
      IF
            ( CASE )
                  penult @  do-QUEDUP =
            IF                                  \ ?DUP 0= IF|UNTIL|WHILE
                  -2 codes  gap
                  do-QUEDUP-BRANCH-UNLESS-ZERO  COMPILE,
            ELSE
                  penult @  do-ZERO-EQUAL =
            IF                                  \ 0= 0= IF|UNTIL|WHILE
                  -2 codes  gap
                  do-BRANCH-IF-ZERO  COMPILE,
            ELSE                                \ 0= IF|UNTIL|WHILE
                  -1 codes  gap
                  do-BRANCH-UNLESS-ZERO  COMPILE,
            ( 0 ENDCASE ) THEN THEN
      ELSE
            last @  do-DUP =
      IF                                        \ DUP IF|UNTIL|WHILE
            -1 codes  gap
            do-DUP-BRANCH-IF-ZERO  COMPILE,
      ELSE
            last @  do-LITERAL =
            ANDIF
                  next  1 codes -  @ 0=
            THEN
      IF                                        \ 0 IF|UNTIL|WHILE
            -2 codes  gap
            do-BRANCH  COMPILE,
      ELSE                                      \ IF|UNTIL|WHILE
            do-BRANCH-IF-ZERO  COMPILE,
      ( 0 ENDCASE ) THEN THEN THEN
;
```

We will optimize other things in ?branch,, but they will all follow that pattern. n < and n >, for a literal n, can be optimized with the following paradigm.

```
: <
      ( CASE )
            STATE @  0=
      IF
            do-LESS  EXECUTE
      ELSE
            last @  do-LITERAL =
      IF
            ( CASE )
                  next  1 codes -  @ 0=
            IF                                  \ 0 <
                  penult @  last  !
                  -2 codes  gap
                  do-NEGATIVE  COMPILE,
```

```
            ELSE                                    \ n <
                  penult @  last  !
                  next  1 codes -  @
                        -2 codes  gap
                        do-LITERAL-LESS  COMPILE,
                        ,
            ( 0 ENDCASE ) THEN
      ELSE                                          \ <
            do-LESS  COMPILE,
      ( 0 ENDCASE ) THEN THEN
; IMMEDIATE
```

For =, we want more optimizations.

```
( Optimize:  DUP n =    0 =    n =    n OVER =   )

: =
      ( CASE )
            STATE @  0=
      IF
            do-EQUAL  EXECUTE
      ELSE
            last @  do-LITERAL =
      IF
            ( CASE )
                  penult @  do-DUP =
            IF                                      \ DUP n =
                  0  last  !
                  next  1 codes -  @
                        -3 codes  gap
                        do-DUP-LITERAL-EQUAL  COMPILE,
                        ,
            ELSE
                  next  1 codes -  @ 0=
            IF                                      \  0 =
                  penult @  last  !
                  -2 codes  gap
                  do-ZERO-EQUAL  COMPILE,
            ELSE                                    \ n =
                  penult @  last  !
                  next  1 codes -  @
                        -2 codes  gap
                        do-LITERAL-EQUAL  COMPILE,
                        ,
            ( 0 ENDCASE ) THEN THEN
      ELSE
            last @  do-OVER =
            ANDIF    penult @  do-LITERAL =   THEN
      IF                                            \ n OVER =
            0  last  !
            next  2 codes -  @
                  -3 codes  gap
                  do-DUP-LITERAL-EQUAL  COMPILE,
                  ,
      ELSE                                          \ =
            do-EQUAL  COMPILE,
```

```
      ( 0 ENDCASE ) THEN THEN THEN
; IMMEDIATE
```

The arithmetic operators should all receive optimization.

```
( Optimize:   x n +    x + n +    0 +    n +    )

: +
      ( CASE )
            STATE @ 0=
      IF
            do-PLUS  EXECUTE
      ELSE
            last @  do-LITERAL =
      IF
            ( CASE )
                  penult @  do-LITERAL =
                  ORIF    penult @  do-LITERAL-PLUS =    THEN
            IF                                  \ x n +   or   x + n +
                  0  last  !
                  next  1 codes -  @  next  3 codes -  @  +
                        -4 codes  gap
                        DUP  ORIF    penult @  do-LITERAL =    THEN
                        IF
                              penult @  COMPILE,
                              ,
                        ELSE    DROP    THEN
            ELSE
                  next  1 codes -  @ 0=
            IF                                  \ 0 +
                  penult @  last  !
                  0  penult  !
                  -2 codes  gap
            ELSE                                \ n +
                  penult @  last  !
                  next  1 codes -  @
                        -2 codes  gap
                        do-LITERAL-PLUS  COMPILE,
                        ,
            ( 0 ENDCASE ) THEN THEN
      ELSE                                      \ +
            do-PLUS  COMPILE,
      ( 0 ENDCASE ) THEN THEN
; IMMEDIATE
```

Pinhole optimization, like any form of peephole optimization, should work across uses of POSTPONE. Given the definition,

```
: UNLESS    0 POSTPONE LITERAL    POSTPONE =    POSTPONE IF ; IMMEDIATE
```

when UNLESS is used, POSTPONE = should combine with the result of the preceding 0 POSTPONE LITERAL to yield 0=, and then POSTPONE IF should yield the BRANCH-UNLESS-ZERO optimization.

(Fast Forthward, from page 38.)
For linear expanses of homogeneously subdivided code, standard editor tools are sufficient.

Another way involves heterogeneous subdivisions, such as header files as opposed to source files. These more diverse units of code create a demand for more refined tools for managing source code beyond editors. Diverse units of code bring opportunities for tools such as make utilities and code browsers.

Diverse ways of structuring code is a catalyst for better code-manipulation tools, accelerating the evolution of development environments. As one example, class hierarchies provide impetus to apply graphical or outline metaphors to the presentation of code.

Code browsers and related tools typically must parse through source code before its compilation. That way, even before compilation, a code browser can be responsive to the code's organizational divisions, such as classes, subclasses, messages, methods (functions), and data structures.

A Time and Place for a Command Interface

Even though I usually tout visual tools, I like the idea of demand-based browsing of source code. That means I don't have to locate files and scroll buttons to see my code. The fewer hoops to jump through, the better.

Forth's customary `VIEW` or `SEE` provisions are demand-driven ways to look at just the code we want to see. However, I suppose their ever-so-incremental nature and limited reach (compiled words only) make them poor candidates to fulfill all our code browsing needs.

Still, I'll bet there is a way to dynamically compute a virtual browsing sequence, such as one that consolidates the source code for the words from one vocabulary as if it were a linear expanse of code. Related challenges are computing these browse sequences before compilation occurs (industrial-strength preprocessing?) and storing any changes back to the true location of the source code (recalled how?).

I suspect this is an area of great opportunity.

Reaching the Limits of Vocabularies

Forth possesses code subdivisions that are fairly mundane, considering that they are mostly homogeneous. The dictionary is an array of definitions, each made up of an array of words.

Vocabularies are a different kind of creature, however. Perhaps they can be seen as a refinement of the dictionary data structure itself. In any case, if we judge them based upon their effectiveness as tools for organizing source code, they don't measure up.

A Forth vocabulary is an impractical means of organizing source code because of a very annoying outside hindrance: Words residing in the same vocabulary are haphazardly organized, due to the compiler's requirement for code to be encountered in a sequence that satisfies load-order dependencies. The final result is that the compiled code for the words in a given vocabulary is strewn throughout the dictionary (held together as a group by the associated chain of link pointers); and the source code for those same words is dispersed throughout our source code file(s).

So, at least in terms of its physical layout, our code is typically not more organized due to its thoughtful placement in vocabularies. Vocabularies as we know them today have another role to play—resolving search-order problems arising due to name collisions.

Nevertheless, once code is compiled, enhanced browsing can be supported by exploiting the extra diversity introduced by vocabularies: An outline view of the dictionary can be supported that is vocabulary-disciplined. A toggle could allow the outline to be restricted to only those words in the current search order.

This helps reveal that vocabularies perform a virtual resorting of code. Vocabulary tools link compiled words in accordance with their vocabulary affiliation, while the source can be ordered distinctly differently.

Notice how easy it is to get carried away with these thought experiments: While we're viewing the dictionary in an outline form, a drag-and-drop interface for movement of words between vocabularies could be a nice accompanying touch. Implementation-wise, however, such a change to a word's vocabulary affiliation would not be an easy one to propagate back to the source code.

A New Direction to Take Forth

The chief culprit that eclipses our attempts to better organize our application code is the banal requirement of having to satisfy load-order dependencies. So despite all the freedoms Forth has brought us, it has not given us the freedom to organize code as well as could be desired.

With C, a similar requirement is imposed, but a means has been provided to circumvent it. This is a means that involves no additional overhead. In C, interface declarations (function prototypes) can hold the place for a routine whose definition you don't want to supply near the location where the compiler needs to be able to recognize the associated symbol.

These placeholders are easy to supply near the point of need. The point of need is before references to any as-yet-undefined, or externally defined, functions. However, any point earlier than that will work equally well. In a header-file fashion, you are free to provide a complete list of function prototypes (interface declarations) as a preamble to your code. In so doing, you gain the freedom to define functions in whatever order suits you (and using whatever file subdivisions suits you).

Often, this practice is not enlisted wholesale, but only when a conflict arises due to the compiler's requirements and the programmer's preference. So sporadic use of this technique permits the C programmer to come out as the winner of these periodic struggles.

(At least one C text* advises the wholesale approach to this function-prototype-before-definition rule. It offers some simple code for header-file-generating tools based upon UNIX awk scripts.)

Deferring words is Forth's equivalent technique, but it introduces overhead that C's separate "interface declara-

**Portable C and UNIX System Programming*, J.E. Lapin (Prentice-Hall, 1987)

tion" technique does not. Furthermore, the use of C function prototypes serves multiple purposes, including allowing "safe" references to precompiled functions inside of libraries, allowing references to as-yet-uncompiled functions for which definitions appear in another file, or allowing forward references to as-yet-uncompiled functions that are defined at a more distant place in the same file.

For Forth's future evolution, we should seek a new provision that can take us out of the straightjacket in which we find ourselves in terms of flexibility of organization of source code. Vocabularies are not the answer, because they act more as organizers in the domain of compiled code than in the domain of source code.

Files

Part of the ingenious simplicity of Forth is its blurring of the ordinarily strong distinction between data structures and procedures. Nevertheless, who wants to confront a large Forth application treated as an unbroken linear expanse of definition after definition?

Certainly, we can and should support files. Many Forth systems that have files also permit us to have blocks inside files.

While files do not overcome the hindrance of ordering our code according to the dictates of load-order dependencies, they are an important way to maintain clusters of definitions.

Files and blocks are among the few organizational tools that we can deploy on the side of source code, while vocabularies can help organize things on the side of compiled code.

Managing Code with Library Protocols

Library archive files support a protocol through which a conventional (C) compiler admits a routine into an application automatically upon determining the call for it in a particular application.

By permitting the programmer to refrain from duplicating reused code in the files and directories that house the source code of several applications, a significant maintenance burden is avoided. The shared source code can be maintained in just one place. (This also lends support for mutable code bases, which I described in the last "Fast Forthward" installment.)

Furthermore: (1) the calculation of the load-order is still automated; (2) the function prototypes in the header files bring automatic interface-checking; and (3) the header files also bring the freedom to reference the library functions at any point within any file that needs to reference them. It's hard to imagine a more complete solution than this!

Through conditional compilation or interpolation of files, we can approximate such a library protocol. However, relying upon preprocessing provisions to achieve this goal produces a cumbersome solution.

Contrast this with the user interface for a C library archive or a C++ class hierarchy: These tools allow code to be incorporated into an application on a demand basis alone. No distracting conditional compilation directives need intrude their way into the code. The code just needs to be written normally.

Nevertheless, if mixing-and-matching bits of code from several code bases is what you need to do today, elaborate text-interpreter processing is one way to achieve the goal (see the previous installment for a related discussion). We already have made headway in this direction because of words like INCLUDE in the ANS Forth standard.

To be more innovative, we can start looking for solutions more on a par with those of C and C++.

Consider an abstract class (or class library). The abstractness of an abstract class derives from the fact that we are not normally permitted to instantiate an abstract class. We can only inherit from it.

This is a form of library-like protocol, because if the application fails to inherit from an abstract class, the application does not need to engage any of its functions. Upon detecting this, the compiler can remove the routines and data structures of the abstract class from the application. (Because it never inserts them, their removal is a matter of doing nothing.)

So a proliferation of abstract classes should be able to encapsulate a library-style protocol for code reuse. This protocol would exploit one of the simplest possible user interfaces conceivable: "reference it or lose it."

Progress Marches On

I appreciate the ease with which I can create both character and paragraph styles in my favorite word processor. Ultimately, it saves me work to delimit text with both types of styling provisions. Despite the sophisticated interleaving of these two distinct styling provisions, I am able to easily anticipate and obtain the formatting I desire. This illustrates to me how a richer palette of formatting units creates more powerful word processors.

Rich partitioning of source code can lead to a variety of sought-after benefits: safer code reuse, better code encapsulation, and better provisions for code reuse (a.k.a. inheritance or template provisions). Greater code clarity brought about through its richer delimiting is significant, too.

Unfortunately, as qualitatively different units of code are interleaved to gain all of these valuable benefits, the added complexity can be daunting. For example, C++ went way overboard.

Signs of relief have appeared, however, indicating that C++ has exacerbated the complexities unnecessarily.

Judging from what I've been reading about Borland's release of Delphi, the merits of diversely partitioned code can be delivered in a much simpler language and programming environment. However, seeing how I have already consumed a considerable quantity of column-inches, I'll leave a discussion of certain Object Pascal wins to a future gathering of the "Fast Forthward" kind.

Before I go, however, I can't resist repeating this statement from Larry Constantine, which he made in his "Peopleware" column in *Software Development* (June 1995):

> Forget the hype of the true believers who tell you it's a new paradigm for thinking about problems; it's all about better packaging. Classes, which are the essential components of object-oriented programming, are just better containers for code.

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

Organizing Code—Hindrances and Aids

Mike Elola
San Jose, California

The partitioning of Forth source code into lines can be an arbitrary business. For improved readability, we devise line-break conventions. With these conventions to help compel us to write code more consistently, we are better equipped to improve the reading comprehension of those who study our code.

Imagine the difficulty of comprehending written language if it lacked subdivisions. Continuous expanses of text with no sentence, paragraph, or section endings would seriously hinder our comprehension of it.

For lengthy documents, we employ still other special conventions, such as numbered paragraphs or sections to help establish smaller frames of reference.

For lengthy listings of program code, we need similar conventions.

Blocks and files can create frames of reference by their attachment of a number or a name to an expanse of code. In object-oriented languages, class names and class hierarchies help chop up the application and make it more manageable.

Vocabularies act more as organizers in the domain of compiled code than in the domain of source code.

Like files, classes are of arbitrary length. Classes also merit attention because the object-oriented languages can understand and can meaningfully process a class name as a scoping mechanism. In contrast, line numbers, blocks, and file names play a role closer to bookmarks.

Furthermore, class hierarchies can organize program code in gross as well as refined ways.

Responsiveness to Units of Code

Consider how well hierarchies of file-system folders can organize our files. Consider how outlining modes are increasingly supported by popular word processors.

A hierarchical or outline organization is perhaps the best means we have to organize information, so why not also apply such a structure to the source code within a file? This suggests viewing our source code as outlines or charts.

If we overlook more granular subdivisions of code, a huge opportunity is missed. In the world of object languages, the equivalent capability takes the form of "code browsers."

While object-oriented languages are sensitized to more granular code units, conventional languages lag far behind. Specifically, when preprocessors unravel the code from multiple files into a single continuous stream, the compiler (the programming language) misses the opportunity to be able to respond appropriately.

For C, sensitivity to file-level scope is more a linker function than a language function. This is not the case when the "external" keyword gives notice to the compiler that a nearby symbol is not defined in the same file. Even in the presence of such cues, however, C does not know or even care to determine which file ultimately defines the symbol. That task is deferred to the linking step, which I consider a language-independent processing step.

In C's favor, the compiler requires knowledge about the interface, if not the actual definition of externally defined symbols. However, this once again requires the intervention of programmers to provide the necessary clues in the form of function prototypes. These clues are provided in header files which, in accompaniment with the C preprocessor, dispense knowledge to the compiler of as-yet-undefined symbols or previously compiled functions that reside in a library archive.

The use of a separate declaration of the function interface and the function definition may create an added maintenance burden. However, this was also a stroke of genius on the part of C's designers. Interface-checking remains an important feature of both C and C++.

The Catalyst for Browsers

Witness that, for conventional compiled languages, several types of files are common. The trend is for code to be partitioned in a rich variety of ways, most of which can be categorized in one of two ways:

One way involves homogeneous subdivisions, such as statement-oriented syntax units that are repeated often.

(Continues on page 36.)

Backspace

Readers are encouraged to respond to "Fast Forthward." Here, Doug Philips probes the preceding installment ("For Want of a Kernel Development Environment").

"Architectural Support for Kernel Extension" is a *great* idea, especially the framework for hanging new functionality. However, as you noted, ANS Forth backed away from, or didn't even try to approach, the "development environment." I suspect that it will be very tough to come up with a development environment/Forth system which is extensible, yet does not specify implementation details at too fine a level. I am not even sure yet that a kernel extension environment is in any way different from a specified implementation.

I must point out that Mac-based Forth systems have been around for a long time, so I take you task for assuming that any kind of GUI enhancements are tied to Windows. If I were to be a Mac snob, I might notice that the Mac has been doing GUI for a lot longer than Windows, and that if Windows needs GUI refinement or interface enhancement, it is because it is so much more immature. But I digress. I just wanted to point out a Windows-centrism that I think is irrelevant to the points you are making in this article. Back on track...

I am most puzzled by your scorn of metacompilation. There have been several articles in *FD* alone, over the past few years, that have tried to debunk the myth that "oooh, metacompilation is *hard*." Clearly, they have failed. One thing that remains puzzling, though, is why you don't consider simplifying or regularizing metacompilation just like you want to do with the Forth kernel itself. Instead, you are advocating (if I read you correctly) that C become the assembly language for Forth, and that metacompilation be replaced by ordinary C compilation. I don't see this as a particularly attractive tradeoff, though I once thought it was. The advantage to it seems to lie in writing the Forth kernel in as common a dialect of C as you can, to enhance portability. However, using C means having a different language that someone needs to know in order to make certain kinds of kernel enhancements, or to deal with C compiler portability issues, or to depend on having a C compiler (or cross-compiler) available to get to a new platform. Attractive as C is as a "universal" assembler, both in terms of "I already know C" and "C is everywhere, writing a C compiler is someone else's problem, and besides there are C compilers for every platform that *I* care about," I think it is the wrong approach. As to being able to link with C code, yes, I think that is very important. Again, I am worried about being seduced by the apparent ease of doing so simply because linking C with C is easy.

Why not address the problem of being "linker friendly"? On many Unix systems, one can link C, Fortran, Pascal, etc. together in the manner you describe. In fact, once you have a library file, it is completely irrelevant what language it was built from. Again, I don't think having a component-oriented Forth requires building Forth from C, though that is a very attractive and "it can be done now" possibility. In fact, it might even be worth doing as a step along the path to getting a real metacompilable Forth that is no harder to understand than simple Unix makefile technology. What bothers me about it is that it would not be an absolutely trivial effort and I am not convinced that it really would be a step in the metacompilation-knowledge-enhancing direction—at least not a big enough step to make the effort worthwhile.

On page 37, near the top of the first column on that page, you say: "But what if some speedier or more expansive file I/O hooks into the kernel are needed? Then you may be forced to undertake an overhaul of the kernel, where you don't have the clarity and other advantages of a high-level programming environment." This has me completely baffled! It makes sense *only* if you consider the alternative to C to be assembly language (and not metacompilation); it also assumes the assembly language implementation has somehow lost the component-ness you were giving to it earlier. Furthermore, I think you are making assumptions which are just not reasonable. The danger/downside of an assembly language kernel is that the assembly language is at too low a level. You never raise the consideration that what is needed can't be done. However, that is a much greater danger when you have to write your kernel on top of C. What if you need to have functionality changes, not just at the Forth kernel level, but also at the component level? This happens a lot with C, in that you need to plug in your own memory-management functions. But the C library comes with its own memory-management functions and, even worse, other functions in the C library that you want to use will be using the C library's memory-management functions. Boom, you are hosed. Many, many times, the only solution is to create yet another layer of indirection (my_malloc, my_free) which can be hidden behind a clever layer of macros, but that in turn has *its* own problems! Yes, you might argue that the problem is that the C library is too monolithic, and that C compiler/runtime vendors ought not to bundle it that way. But they do. And while you might find more component-oriented systems (GNU C *might* be amenable to being modified in this way, though maybe not... at least you can get the source code to experiment with!), that drastically reduces the portability of the code that depends on it. Such a reduction would completely undermine the portability justification for using C.

I don't see this as a particularly attractive tradeoff, though I once thought it was.

Further down that same page, you say, "Using lots of execution vectors in a Forth kernel gives us a somewhat similar flexible code base in a Forth environment." I would assert that they give you exactly the same flexibility. The mild performance hit isn't a matter of flexibility, but merely a side effect of how it is implemented. In fact, it gives you an even greater amount of flexibility, because there is no widespread ability in C to revector function calls on the fly. Yes, you can use function pointers in C, but they are exactly the same as DEFERed words. The "plug and play" linking process resolves the symbols once and for all. The difference here is that, in Forth, the user of the word can benefit from using a DEFERed without knowing it is deferred, whereas C requires you to write function calls through function pointers differently... you can't merely change the definition of the

(Continues on page 30.)

CALL FOR PAPERS
FORML CONFERENCE

The original technical conference for professional Forth programmers and users.

Seventeenth annual FORML Forth Modification Laboratory Conference
Following Thanksgiving November 24–26, 1995

Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California USA

Theme: Forth as a Tool for Scientific Applications

Papers are invited that address relevant issues in the development and use of Forth in scientific applications, processing, and analysis. Additionally, papers describing successful Forth project case histories are of particular interest. Papers about other Forth topics are also welcome.

Mail abstract(s) of approximately 100 words by October 1, 1995 to FORML, PO Box 2154, Oakland, CA 94621. Completed papers are due November 1, 1995.

The Asilomar Conference Center combines excellent meeting and comfortable living accommodations with secluded forests on a Pacific Ocean beach. Registration includes use of conference facilities, deluxe rooms, meals, and nightly wine and cheese parties.

Skip Carter, Conference Chairman

Robert Reiling, Conference Director

Advance Registration Required • Call FIG Today 510-893-6784

Registration fee for conference attendees includes conference registration, coffee breaks, and notebook of papers submitted, and for everyone rooms Friday and Saturday, all meals including lunch Friday through lunch Sunday, wine and cheese parties Friday and Saturday nights, and use of Asilomar facilities.

Conference attendee in double room—$395 • Non-conference guest in same room—$280 • Children under 18 years old in same room—$180 • Infants under 2 years old in same room—free • Conference attendee in single room—$525

Forth Interest Group members and their guests are eligible for a ten percent discount on registration fees.

Registration and membership information available by calling, fax or writing to:

Forth Interest Group, PO Box 2154, Oakland, CA 94621, (510) 893-6784, fax (510) 535-1295

Conference sponsored by the Forth Modification Laboratory, an activity of the Forth Interest Group.