$10 Volume XV, Number 2 July 1993 August

FORTH DIMENSIONS

The FSAT Project

Mixed Integer Arithmetic

One-Screen Virtual Dictionary

H-P Deskjet/Laserjet Driver

Natural Language Programming

Contents

Features

Departments

Editorial

What is the ideal Forth development environment? Opinions range from bare-bones fig-Forth (simple, maybe elegant) to F-PC (rich) to hardware implementations (fast), graphical interfaces (contemporary), object oriented (post-modern?), etc. Of course, how you define a development environment will depend on whether you are looking at the issue from within the Forth community, the larger software community, or the world of embedded systems.

Whatever your perspective, you still have time to share your expertise and opinions *and* win up to $500. The deadline for entries in the *Forth Dimensions* "Forth Development Environments" contest is August 1, 1993. Completed papers (hard copy and diskette, please) must be received at our office by that date. The theme is the same as for the upcoming FORML conference, and papers for that event are also eligible for the *FD* contest *if they are received by the contest deadline;* just mail a separate copy to the attention of the *FD* editor. For some of the suggested topics, see the FORML announcement on the back cover; contest details are provided in the ad on page six of our last issue.

* * *

Our last issue carried Donald Kenney's "Forth in Search of a Job." The author gave reasons why a software manager might reject Forth as a programming solution in a corporate environment. The objections sounded familiar and a little depressing (perhaps even reasonable, from a certain point of view). Were they blunt words, a death knell, or a healthy dose of reality therapy? Well, old hands at the sales game know that once you have a clear picture of a prospective customer's objections, you are on the road to making a sale.

Elizabeth Rather, President of Forth, Inc. and chairperson of the committee developing ANS Forth, responds in this issue with a letter I'd like every Forth programmer and business to carry prominently in their consciousness. After twenty years of running a leading Forth business, Ms. Rather has heard all the arguments against Forth and has an impressive arsenal of evidence and case histories with which to shoot down each of those objections. We are fortunate to have her no-nonsense attitude and get-the-job-done approach in our community.

It seems to me that, as always, we have choices about how we react to the world. We can beat our breasts and whine about Forth's market share, or we can use our assets to penetrate and impress the portion of the marketplace to which we do have access. We can place a couple of ads about our product or service and wait for something (or nothing) to happen, or we can prepare ourselves to go out and meet our prospects face to face. We can cringe at objections to Forth and slink back to our offices to lick our wounded egos, or we can counter those objections with hard facts and real examples, and then demonstrate the validity of our claims, proving both our own expertise and Forth's facility by doing a great job and, thus, creating another happy client.

Unlike Michaelangelo, the Pope is not elevating us to the chapel ceiling; if our art is important to us, we will erect our own scaffolding, one success at a time. Sure, we'd all like to ride the coattails of an endorsement by Bell Labs, for example. But instead, we must rely on business acumen, including sales techniques, as a necessary complement to our programming skills.

* * *

We also present in this issue a new author. Jim Schneider's FSAT project is of considerable scope, tackling the thorny and controversial issues involved in providing a Unix-like Forth development environment which will allow, among other things, compiling C routines into Forth. While progressing to that end, he promises to provide some interesting utilities to keep us happily occupied while he sweats the details of the big picture. And, for the sake of those who repeatedly ask, "Where is the younger generation of Forth programmers," Jim has provided details of his own rites of passage.

Discerning readers will notice congruities between Jim's article and Mike Elola's essay in this installment of "Fast Forthward." I'm tempted to ponder whether this represents Forth's future. But it is probably more accurate to observe that this is another of the faces that Forth's extensibility—malleability—permits it. Whether its virtue suffers or is lost entirely in the translation will depend on both the eye of the beholder and the mind of the implementor.

—Marlin Ouverson

Forth Dimensions
Volume XV, Number 2
July 1993 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, c/o TPI, 1293 Old Mt. View-Alviso Rd., Sunnyvale, CA 94089. Second-class postage paid at San Jose, CA. POSTMASTER: Send address changes to *Forth Dimensions*, P.O. Box 2154, Oakland, CA 94621."

Letters

Letters to the Editor—and to your fellow readers—are always welcome. Respond to articles, describe your latest projects, ask for input, advise the Forth community, or simply share a recent insight. Code is also welcome, but is optional. Letters may be edited for clarity and length. We want to hear from you!

Smashing Marketplace Myths

Dear Mr. Ouverson:

I was appalled by Donald Kenney's article, "Forth in Search of a Job" (*Forth Dimensions* XV/1). I have certainly heard these objections many times, and I agree that it's important for all of us to know that these are the fears that haunt some managers. What I object to is his bland acceptance of these myths, the assumption that these bogeymen are real, despite considerable real-world evidence to the contrary. Let's take them one at a time.

1. *Integration, maintenance, bug fixes:* As we have been serving professional Forth users for 20 years now, we and our customers have many examples of both large systems (which I'll discuss later) and projects that have evolved over many years. We have found that Forth wears extraordinarily well. In fact, our very first customer, Bob Barnett of Cedar Rapids, Iowa, has developed a successful data entry system over nearly 20 years (we first worked with him in the spring of 1974), on at least six different CPUs. California Municipal Statistics, in San Francisco, has a very complex data base program that Chuck Moore first developed in 1976 that is still in use; they've done a lot to it themselves, and have called us in a few times to assist with special projects, such as porting it to more modern platforms.

   More recently, Federal Express has been maintaining two major Forth-based systems, the oldest of which dates from 1987. They have a team of about eight programmers, who maintain both these and some C-based systems. They have told us they find the Forth-based systems far easier to maintain than the C-based systems; they can turn change requests into releases in only a few weeks rather than months.

   It is specious to equate using Forth in a shop that may also use C to "22 dialects of 13 programming languages."

2. *"Choice of language doesn't matter."* I agree that it often seems that way to most managers, and programmers as well. In regard to choosing, say, C vs. C++ or Pascal, it's probably even accurate. But choosing Forth can often reduce costs as well as calendar time (which is often more important) by really significant margins. Not only can the work go quicker, with a smaller programming team, it frequently also requires much simpler, cheaper computers to run on.

   When we developed the Heating, Ventilation, and Air Conditioning (HVAC) system for the GM/Saturn plant in Tennessee, they told us we saved them several *million* dollars over more conventional approaches. We did this by developing software that ran on cheap, embedded Z-80 boards coordinated by a few PCs rather than using many VAXs (as proposed by the next lowest bidder). On the Saudi Arabian airport project in the mid-1980's, we produced in 18 months and about 30 programmer-years software that a prior team had failed to deliver in three calendar years and 100 programmer-years.

   Of course, we're experts. But a few years ago, Cameron Lowe of Bell Canada wrote in a magazine article that he and his partner completed in three months a project that their C experts had estimated at three to four years—and it was their *first* Forth project.

   These are not trivial differences. Savings of years and tens of thousands or millions of dollars will get most managers' attention.

3. *Cost and availability of Forth programmers:* As presented, the issue is "grabbing a body quick for a short time." In fact, in most communities there exists a group of frustrated Forth programmers who are unhappily programming in C because prospective employers are afraid to use Forth because there aren't any Forth programmers!

   But it is also true that not knowing Forth is a remediable condition. We offer courses here almost every month, or on-site if there are more than three or four people to train. There are other trainers as well, not to mention books and well-documented Forth products that we straightforward to learn. Often companies are better served by finding really sharp programmers (assembler, C, etc.) who are familiar with the application domain in question, and letting them learn Forth.

   As much as I hate to say it, not all Forth programmers are equally qualified for a particular project. If you're doing a time-critical embedded system, training a C programmer familiar with the processor and application in Forth will probably produce a better result than using some-

Forth hasn't taken over the world—not for any technical reason, but for a marketing reason.

body whose experience is limited to PCs and high-level Forth. In fact, a good C or assembler programmer trained in Forth and using Forth will very likely do better than the same person using C! (Re-read the item about Cameron Lowe, above—we hear many such stories.)

When I talk with folks running the larger Forth shops, I occasionally ask whether programmer recruitment has been a problem. Although they would all rather have more top-notch Forth programmers besieging them with resumes, they generally have no serious complaints.

As to cost, the important thing to remember is the overall cost of getting the project done, not the hourly fee for the programmer! What you want is the best-qualified person, who will do the best job for the best overall price. We have long, happy relationships with people all over the country who have found it cost effective to deal with us for things ranging from small "tweaks" to an existing system, to major projects. For a number of years, for example, we've supported Bob Gherz, an astronomer at the University of Minnesota, who has a polyFORTH-based telescope control program that he runs at three different observatories. He can call us on the phone, ask for some little thing, and we'll likely have it done and transferred by modem in a few hours. Although he has bunches of dirt-cheap graduate students at his beck and call, he says he finds it much less expensive overall to deal with us.

4. *The suits in the front office...* might be delighted to hear you've found a way to save them significant time and money by using a technique that's been used successfully by many of the technology leaders of America. No fairy tales needed.

5. *"The system design will probably have been bungled."* Yes, sigh, it usually is. But even a lot of Forth detractors admit that it's great for developing by "successive prototyping," the fall-back path of choice when the design is fuzzy or non-existent. Often it's easier to do a prototype of, say, a user interface, than to get somebody to try to imagine on paper how it should look. And when the design decisions are being reconsidered, Forth's interactivity and flexibility really shine.

   As to the large project issue, we've been involved in numerous 5–20 programmer teams over the years. The main difference between these and similar projects using other languages is that each programmer is so productive that you have to be on your toes making sure they're being productive in the right directions. Communication and coordination is even more important (and there is never enough of it in any project!). But is this a liability? Somehow I've never heard managers wishing their programmers were slower or less productive!

   Also, since you can do a particular job quicker and with a smaller team in Forth, some team-management issues solve themselves.

   Readability is an issue in every language, as are other subjective style issues. We have had great success by establishing in-house coding standards that everyone follows. As a result, you can't look at any of our code and guess who wrote it—and programmers never feel the discomfort of having to cope with someone else's idiosyncratic style. Other successful large shops do the same. It matters less what those styles are than that they exist and are observed.

   Shared data and other technical issues are handled pretty much the same in Forth as in other languages. The details are all different, but the principles are the same.

6. *The controls won't adapt:* My experience is that procedures simplify. We have developed some tools for configuration management—as have other shops—that we are reasonably comfortable with, although there's always room for improvement. Federal Express is pretty formal, and report their procedures adapted to Forth well. Sun Microsystems (which uses Forth in the Open Boot on every SPARC Station) uses the UNIX tools.

The basic problem here is that Mr. Kenney is swallowing these common myths whole, unexamined and untested. They make great rationalizations, but there's too much contradictory evidence to respect them as facts. It's a disservice to the Forth community to pass on this stuff without at least checking with some of the folks who have been very successful doing just those things the myths say you can't do.

So where do I think Forth is going? During the first decade of FORTH, Inc.'s life, "everyone" was using Fortran and saw no need for a "new mainline language." Then "everyone" used Pascal (briefly), now C. But look! There on the horizon! It's C++! The fact is that language (like other) fashions come and go. "Mainline" languages were designed for a particular purpose, and then got popular and started being used for other purposes. Forth was designed for a particular purpose: to maximize both programmer and code efficiency, particularly in real-time applications. Lots of people still want to do that. Forth hasn't taken over the world—not for any technical reason, but for a marketing reason: no large, well-known organization has elected to spend a lot of money popularizing it. Until that happens (miracle!), it will continue enabling imaginative companies such as AMTELCO (remember Olaf Meding's article a few issues ago?) to smash their competition and take 80% of the market. Would your company like to do that?

Sincerely yours,
Elizabeth D. Rather, President
FORTH, Inc.
111 N. Sepulveda Boulevard
Manhattan Beach, California 90266-6847

Three-Piece Logic

Dear Mr. Ouverson,

Donald Kenney's article, "Forth in Search of a Job" is, alas, on the mark. At a software conference in February, I noted out of about 250 booths the following language vendors: zero vendors of Forth, Ada, COBOL, Modula2; one vendor of APL, Lisp, Prolog, Smalltalk; two vendors of BASIC, Fortran; three vendors of Pascal; four vendors of C++; and seven vendors of C. In addition, most of the libraries and GUI code generators were in C. The debuggers were mostly geared toward C and C++.

I heard a great story from a programmer with a major software company. He believed C was popular because it behaves like a high-level assembler and is available on many platforms. His job, however, was to write tight assembly code for fast math functions, and then reverse-engineer it into C code that compiles into the original assembly routine. The suits thought this made sense.

Yours truly,
Walter J. Rottenkolber
P.O. Box 1705
Mariposa, California 95338

Potable dpANS Forth

Dear Mr. Ouverson:

In reading Charles Curley's review of Jack Woehr's new book, *Forth: The New Model*, I found myself wondering what version of dpANS Forth Mr. Curley is using (the current version is dpANS55; by July there will be a new and possibly final draft). Over the last year or so, many hours have been spent clarifying issues such as those he raises. Although he is correct that it is not the place of a standard to serve as a tutorial for complete newcomers to the language, leaving the field open for helpful texts such as Jack's, we have made strenuous efforts to make requirements clear. If they are not clear, we would greatly appreciate knowing what is confusing so we can clarify it. But such comments are useful only in reference to a particular draft, preferably the current one, so we can avoid revisiting problems we've already solved!

Consider the example that Curley raises, the definition:

```
: LEMME-OUT
  100000 0 DO
  I 1000 = IF  LEAVE THEN
  I .  LOOP ;
```

In the first place (a point Curley misses), although this may compile fine on Vesta's 68332 32-bit Forth, the standard does not guarantee that the literal 100000 will compile correctly on 16-bit systems! The phrase `ENVIRONMENT? MAX-N` will return the value of the largest usable signed integer on a particular system.

Moving on to the points he does raise, I wonder why he thinks it might not compile? He seems to be concerned about the control-flow stack; however, for each word that has prescribed compile-time behavior, use of the control-flow stack is explicitly documented. In this example, the affected words show their compile-time stack behavior as:

`DO`	(C: -- *do-sys*)
`IF`	(C: -- *orig*)
`THEN`	(C: *orig* --)
`LOOP`	(C: *do-sys* --)

No other words, including `I` and `LEAVE`, have any compile-time action or affect the control-flow stack (which exists only at compile time) in any way. Therefore, if the structures are nested properly (they are), the definition will compile properly. No control-flow stack manipulators are needed; they rarely are.

Curley seems confused by the fact that the *execution* behaviors of `DO`, `I`, `LEAVE`, and `LOOP` have a return-stack notation:

`DO`	(R: -- *loop-sys*)
`I`	(R: *loop-sys* -- *loop-sys*)
`LEAVE`	(R: *loop-sys* --)
`LOOP`	(R: *loop-sys* --)

This relates to the availability of the *run-time* loop control parameters. Note the description of the execution behavior of `LEAVE`:

> Discard the current loop control parameters. An ambiguous condition exists if they are unavailable. Continue execution immediately following the innermost syntactically enclosing `DO ... LOOP` or `DO ... +LOOP`.

Now, admittedly there are a bunch of syllables in "syntactically enclosing." So if you still don't know whether that last `I .` will be executed, check out the Appendix A.6.1.1760 `LEAVE` (many sections have identically numbered Appendix A sections, another good place to keep a tentacle), where it says:

> Note that `LEAVE` immediately exits the loop. No words following `LEAVE` within the loop will be executed. Typical use:
> `: X ... DO ... IF ... LEAVE THEN  LOOP ... ;`

These appendix notes are explanatory only, and do not contain requirements. We supplied them where either we or the many people that have contributed helpful comments (formal or informal) over the years thought extra clarification would be useful.

There certainly is a place for good books providing tutorials, background materials, code examples, tricks and techniques, and many other things that it would be inappropriate to include in a standard. I hope Jack's book is the first of many. But I do believe readers will find the current draft better than Houston Shipping Channel water. It may not be Perrier, but it's at least L.A. Tap.

Sincerely yours,
Elizabeth D. Rather
[Ms. Rather is the chairperson of the ANSI Technical Commit-

tee developing the ANS Forth. —Ed.]

C Saws, Frogs, and Princesses

Dear Marlin,

It is with mixed feelings that I renew my membership for yet another year. Please let me explain...

I am not a programmer, but instead an engineer who must program at least enough to make my hardware designs function. I design for two basically quite different realms: one is interface cards in a PC-type machine that usually includes a user interface, and the other is embedded control processors usually with no user interface. Over the years, for the PC (and before that for CP/M 8080 computers) I have written in assembly or Forth.

With the exception of one sad, sorry experience with Forth on a '196, I usually program the particular embedded processor in its assembler. A couple of years ago, I was dragged kicking and screaming to C because (1) in my work, more and more I was (am) required to be able to look at other people's C code and understand at least a little of it, and (2) I have a commitment as a working engineer to be at least minimally competent is such a widely accepted language. I chose Turbo C 2.0 because of the (really great for a beginner) "user environment." A bonus for me, and the thing that really prompted this letter, is the very powerful graphical functions (BGI).

One project I am currently working on involves upgrading the control system on an industrial machine, and part of the upgrade is to replace about a dozen analog panel meters with virtual instruments on a CRT. The customer wants a PC-type computer for reasons of long-term availability of spare parts. My natural urge to write the code in Forth was squelched by two things: (1) I could not, in good conscience, leave them with code controlling a very expensive machine with very expensive down time that would be essentially unsupportable (they have no internal computer expertise, I am not going to be around forever—I'm an old guy now—and finding someone to understand both Forth and their machine is not likely), and (2) no Forth I know of has a graphics library of panel meters, annunciators, bar graphs, etc.

There is no way that, in my lifetime, I could write in Forth the graphical words to draw these instruments and update them in real time. For $250, I bought a complete real-time graphics and measurement control library (that uses Turbo C's BGI) that lets me easily call functions that draw and update needle and arc meters; scrolling, sweep, and x,y charts; bar graphs; annunciators; and much more.

So the real world compels that I write in C. Do I like C? Not a whole bunch. Almost every line of code is a struggle. I find it very powerful, I like the user interface, I like the hundreds (thousands?) of functions available. Last month, I bought for less than $50 a commercial, supported library that has dozens of functions for user interface (windows, buttons, pull-down and pop-up windows, etc.) that, of course, is compatible with my Turbo C.

What I really miss from Forth is the ability to execute words from the keyboard as an aid to hardware checkout and debug. The edit/compile/link/execute sequence, even in the smooth one-key turbo environment, is clunky compared to the slick, quick, smooth environment I have with Fifth, with its instant compile that occurs when exiting the built-in editor (remember Fifth? it's an F83 variant with a very innovative and elegant coding environment, local variables, a great word-by-word trace feature, and more).

Do I still program in Forth (Fifth)? Yes, of course. There are times when it is perfect. A few days ago, I needed a quick PC program to check out a quadrature-phase rotary encoder for use as a panel-mounted, operator input device for incrementing/decrementing a variable. In less than an hour's time, I had working code in Fifth.

So, finally, to the reason for renewing my Forth Interest Group membership. Even though I paid for and registered my copy of Fifth, I feel that if I continue to use it, I should acknowledge my appreciation by supporting the Forth community. Which is not to say that I like every Forth... there are some truly lame products out there. I've kissed a lot of frogs and found few princesses. Some of the loudest advertisers have produced the least value. But that's a story for another time.

So, okay, here's the $40.

Sid Knox
Helios Systems
Oxnard, California

The FSAT Project

or, How to Build an Operating Environment, Win Friends, and Influence People, All While Being Politically Incorrect (in a Politically Correct Way...)

Jim Schneider
Sunnyvale, California

This series of articles is my attempt to give back something to a language and a culture that has given so much to me. Essentially, I'm attempting to build a development environment that has the advantages of both Forth and UN*X, and hopefully sidesteps the disadvantages.

In my opinion, the use of Forth as a mainstream language is hampered by the fact that programmers not accustomed to the Forth philosophy are loath to give up the tools with which they are comfortable long enough to learn how to be productive in Forth. Additionally, some of Forth's best tools are very difficult to learn, and even harder to modify. Furthermore, there is Forth's perennial standards debate, various gross and subtle inconsistencies between different implementations of the same standard, and a lack of some tools that a UN*X programmer would take for granted. When these points are taken altogether, Forth appears more and more limited. I hasten to add that this is not to say that Forth has to be limited, or limiting, but that Forth doesn't have the tools integrated into it that most non-Forth programmers expect in a development environment.

My number-one priority is merging the capabilities and strengths of Forth and UN*X.

These observations lead me to an inevitable conclusion: the most extensible (in some senses, the only extensible) computer language in the world needs to be extended again, to include the tools that other programmers have come to expect. In the process, I hope to include tools that Forth programmers will find useful.

The FSAT project (which stands for Forth Systems/ Applications Tools) breaks down into several categories. Since the ultimate goal of the project is to have a complete, UN*X-like operating system, complete with the tools that a UN*X user has come to expect, along with the tools unique to Forth, and some tools that, hopefully, will emerge at the interface, I feel it would be logical to decompose the articles into articles about the tools, and articles about the functioning of the operating system and system internals.

(At this point I feel compelled to explain at least part of the title for this article. When I first described the concept of this series of articles to the editor, he said that some would probably feel that the articles wouldn't be politically correct. My response to this is twofold: the weaselly response is that I'm trying to meld a mainstream OS with Forth in such a way that a person comfortable in either would be comfortable with my work. The other—possibly more accurate—response is that if I were worried about being politically correct, I'd be programming in C.)

In this series of articles, I will attempt to describe the variety of tools that I am using to develop the project. For the core Forth system, I'll be primarily using a macro processing language in tandem with an assembler. Once I have the Forth up and running, I'll be using it to write the device drivers and some of the tools for the Operating System. The one system utility that I will be writing with tools other than Forth is the C-to-Forth compiler.

When the bits settle, the system will include these tools:

- a C-to-Forth compiler
- several utilities to create and manipulate relocatable object modules and object code libraries
- a shameless copy of every FSF (Free Software Foundation, an admirable group) utility

I will be using the UN*X utilities *m4(1)*, *yacc(1)*, and *lex(1)* extensively in the project. (Note: the numeral one in parentheses refers to the section of the UN*X man pages where the description of the programs may be found.) I will introduce the important features of these utilities in a separate article. Remember, however, that this is a series of articles on a Forth operating system, and not a tutorial on UN*X utilities, so the descriptions will be somewhat brief.

When I am done, I hope to achieve a few goals. My number-one priority is the merging of the capabilities and strengths of my two favorite environments: Forth and UN*X. From time to time, the pursuit of this goal will lead to some bizarre choices on my part. Closely paralleling this first priority is the goal to make the system POSIX (Portable Operating System Interface) compliant. Because I am using Forth for a major portion of the development, of course it is reasonable to expect that the finished product will be fully as extensible and mutable as Forth itself. This flexibility will be a recurring theme throughout the system. The user interface is going to be a full implementation of ANS Forth, with

certain extensions to allow an interface to certain structures that are placed off limits in the standard, and other extensions that allow the user to enable certain non-standard behaviors.

On a more concrete level, the Operating System itself will provide standard OS services, including multi-tasking, multi-threaded operations, demand paging, and device-independent I/O services. The Forth model (and, indirectly, the entire OS) is based on indirect-threaded code with separate address spaces for the dictionary, headers, data, and stacks. Although I am writing the OS with portability in mind, my first target platform will be the Intel i386/i486 processor family. The operating system will be isolated from user programs by using the protection features of the platform, and entry to the system through a single entry point will be enforced. On the Intel platform, this will be accomplished by making the OS run at a higher privilege level (CPL 1) than user tasks (CPL 3). The OS procedures themselves will interact through a monitor that has a single entry point, and has a higher privilege level than the OS procedures (CPL 0).

In short, I envision a single-user multi-tasking operating system, with the flavor of UN*X, and the power of Forth and UN*X. I hope to enable programmers familiar with either Forth or C to be immediately productive. My hope is that when the C programmer stops writing code long enough to realize what a pain in the butt it is to write C, he (or she) will be pleasantly surprised to find that they are in an environment where they can be immediately comfortable being more productive (using Forth, of course—I can't see any way a C programmer can gain the benefits of Forth without using Forth).

At this point, I think I should mention some caveats:

This system will be a development system, conforming to my (sometimes quirky) needs. As such, trying to write the Great American Payroll Cruncher on it will probably be a waste of resources. The system will support some security, but a truly dedicated hacker with a single password on the system can probably find a way to get into everything on the system. Because the system uses cooperative multi-tasking, use as a timesharing multi-user operating system is not advised.

As a large part of these articles will discuss the tools I developed to enable me to do this project, I figure it's only fair to start with a few tools. The source code accompanying this article is a short selection of the Forth tools I've built over the years. Some of the tools will be used in the articles and some won't be. The tools that I'm not using in the articles are there because I'm an irrepressible tool builder, and I like to have complete, symmetrical sets of related tools (sets I call orthogonal—the only reason for the existence of C+!, etc., in my toolkit). One toolset, in particular, is a kludge I used trying to manage large libraries of source code (and not very successfully, since it relies on absolute block addresses, and I move source code fairly freely). By the way, I use the UN*X convention of letting the vertical bar (|) stand for "or" in a list of choices. Thus, if a Forth word could return a value and a true flag or a false flag, I would diagram its stack effect like:

```
( -- val \ true | -- false )
```

and if it left either a value or zero, I would diagram it like:

```
( -- val | 0 )
```

Enjoy.

About Jim Schneider...

It has been pointed out to me that some may find my background interesting. Well, you asked for it, Marlin.

I've been programming in Forth for about ten years now, and programming in C for about eight. I consider myself to be an expert in both (i.e., if I were given sufficient time and motivation, I could write a compiler for either). I find it noteworthy that I achieved expert status in Forth without benefit of formal instruction, but I've taken 18 weeks (120 hours) of classes on C. I also consider myself to be fairly well versed in UN*X-like operating systems (after about three years of interaction with them, and 120 hours of instruction). Recently, I started looking into C++, a language in which I expect to receive an additional 120 hours of instruction, and in which I'll probably be somewhat less than competent.

For those of you interested in why I have such diverse exposure, I have only my father to (blame I thank, pick one). For several years, my father would find a really neat way of doing something (programming a computer, selecting music, what have you), and run straight out to tell someone. Since I lived with him at the time, I was something of a captive audience. He would go straight to me, interrupt whatever I was doing, and tell me all about the neat little trick he'd discovered. Since I was usually playing "Caverns of Mars," or doing something equally as important, I resented the interruptions. However, he persevered in his quest, and showed me what `: STAR ASCII * EMIT ;` did. When I replied, "That's nice, but what else can it do?" he handed me a copy of Leo Brodie's *Starting Forth* and told me to go to it. I promptly put the book away, and went back to playing "Caverns of Mars." (This was during the summer I spent more time at the computer than I would spend in my state-mandated secondary school classes for the next three years.)

One day (during the same summer), after I'd spent all morning trying to figure out how to get the computer I was working on (an Atari 800) to divide a 10,000-digit number by a 1,000-digit number (and most of the afternoon, and the preceding night, and...), I noticed I hadn't eaten in, oh, 36 hours or so. I went into the kitchen and fixed something to eat. While en route to my customary eating site, I chanced to glance at the previously mentioned tutorial. Since I was (and still am) in the habit of reading while I eat, I picked it up and thumbed through it. I was struck by the fact that it looked like it would be a lot easier to solve my problem in Forth than in BASIC. I spent the next two weeks figuring out enough about Forth to do it, after which I was hooked. During this time, my father was rapidly acquiring a sizable library of Forth tutorials, manuals, and essays. (He was working in Atari's Coin-Op division, which was very interested in Forth at the time.) Over the next couple of years, I read and reread every thing he had on Forth.

During that couple of years, he was laid off from Atari, went to work for a small company named Buscom (which went under), and went to work for Zilog. Near the end of the couple of years, I was recompiling my Forth kernel about once a week, and using it to hack save files for a game. My father, who had shown so much interest in my mastery of Forth up to that point, came to me one day and said, "Well, nobody uses Forth anymore. Everyone's programming in C now. Here, let me show you how to get C to print out 'Hello, World!'" This was typical of his attitude. As soon as I was able to discuss one of his areas of expertise intelligently, he decided the area wasn't important after all, and tried to make me look at something else. This doesn't just hold in computers: he only listens to opera, Irish folk, and country, because I like rock, classical, and jazz.

Anyway, he interrupted a crucial hack of a game file (I was trying to give one of my characters in a Dungeons & Dragons-type game extremely high abilities) without so much as an "Excuse me." When I asked him if it was possible to get C to do anything more profound than print out lame exclamations, he put on a smug smile, and handed me a copy of K&R (Brian Kernighan's and Dennis Ritchie's *The C Programming Language*). I promptly set it down and went back to hacking OUB save files. Eventually (primarily because of my father's nagging), I looked through the book, and discovered that there was a class of problems that are trivial (or nearly trivial) in C that are difficult in Forth. (Of course, the opposite is more usually true.) I worked with either C or Forth for the next several years (with a four-year intermission for the army), using whichever seemed to make the most sense for a given application at the time. I was also exposed to the UNIX operating system (BSD 4.1) during this time, and discovered that it had lots of interesting games.

When I got back from the army, my father was primarily doing UN*X shell programming for Novell. Since this was the first time I had continuous access to a UN*X-like operating system (he had SCO XENIX on a box at home), I figured I'd give it a whirl. Imagine my surprise when I discovered that it was almost exactly what I'd ask for in a development environment if I were restricted to C. The system is well integrated, and it makes a lot of sense (for a programmer).

Now my poor father, after having shown his son Forth, C, and UNIX, is writing Microsloth Windows apps in C++. Well, Dad, after looking at both Windows and C++, I have to tell you, you're safe. I don't think God Himself could persuade me to take either of them seriously.

Glossary

`&!  ( val \ addr -- )`
Logically AND val with the value at the address addr on the stack. The result is stored at addr. Pronounced "and-store". Included to make +! a member of an orthogonal set.

`-!  ( val \ addr -- )`
The value val is subtracted from the value at the address addr.

`--  ( -- )`
Reads and discards the remainder of the input line. The same as the dpANS standard word \. -- is an immediate word.

`?NUMBER  ( addr -- double \ true | false )`
Attempts to convert the string at addr into a double-precision number in the current base, if possible, and leaves a flag on the stack to signal its success or failure.

`AUTOLOAD  ( n -- )`
If the next word in the input stream is not found in the dictionary, the block number n is loaded. This is an example of a (not very successful) kludge used in managing libraries of source code.

`AUTOLOAD_FROM  ( n -- )`
If the next word in the input stream is not found in the dictionary, the block number n is loaded from the screen file named in the second next word in the input stream. This is an example of another kludge used to manage libraries of source code. Both of these kludges suffer from the fact that blocks can be inserted between existing blocks on most block-oriented Forth file systems (and thus causes references to the wrong blocks).

`C&!  ( val \ addr -- )`
Logically AND the character value val with the value at address addr. This is another function to build a complete set of operators analogous to +!.

`C-!  ( val \ addr -- )`
Subtract the character value val from the value at address addr.

`C+!  ( val \ addr -- )`
Add the character value val to the value at address addr. A character-oriented analog of +!.

`CELL  ( -- n )`
A constant value corresponding to the number of address units in a word. On most 16-bit Forth systems the value is 2, while on most 32-bit systems the value is 4.

`CELL+  ( n -- n' )`
The constant value CELL is added to the item on the top of the stack.

`CELL-  ( n -- n' )`
The constant value CELL is subtracted from the item on the top of the stack. These two words are primarily used for addressing of adjacent fields.

`CELL/  ( n -- n' )`
The constant value CELL is divided into the value on the top of the stack. This word is used primarily to determine the number of cells an object will occupy.

`CELL/MOD  ( n -- rem \ quot )`
An alias for CELL /MOD. Depending upon the implementation, can be faster than the actual code fragment. For example, on a 16-bit system, this can be implemented as DUP 1 AND SWAP 2/, which is faster because 2/ is implemented as a shift, and not a true divide.

`CELLS  ( n -- n' )`
The constant value CELL is multiplied by the value on the top of the stack. Used to determine the number of address units occupied by a multi-cell object, or to access a field n cells from the beginning of a record.

`C^!  ( val \ addr -- )`
Logically XOR the character value val with the value at addr.

`C|!  ( val \ addr -- )`
Logically OR the character value val with the value at addr.

`^!  ( val \ addr )`
The value val is logically XORed with the value at addr.

`c?num  ( double \ addr \ sign \ flag -- double \ true | -- false )`
Cleans up the output of ?NUMBER.

`s?num  ( addr -- double 0 \ addr' \ -1 \ true \ sign flag )`
Sets up the BEGIN...WHILE...REPEAT loop of ?NUMBER.

`wl?num  ( double \ addr -- double \ addr' \ decimal place \ flag )`
The WHILE loop portion of ?NUMBER.

`wt?num  ( double \ addr \ decimal place \ flag -- double' \ addr' \ true | -- double' \ addr' \ [ true | false ] \ false )`
The WHILE test portion of ?NUMBER.

`|!  ( val \ addr -- )`
The value val is logically ORed with the value at addr.

Bibliography and Suggested Reading

Be advised: most of the works I will cite are either documentation for products, or out of print, or documentation for products that are no longer available. If the work is the documentation for something, you may be able to obtain it from the vendor. They will probably sell the documentation for a substantially smaller sum than the actual product. The books that are out of print, or the documentation for unavailable products, will be harder to come by. In most cases, the documentation for the later products will suffice. If this is not the case, I will be sure to point it out. The out-of-print books will probably be found in secondhand bookstores, or by asking small volume publishers to reprint a very few copies.

PC FORTH v1.26 Reference and User Manual
The documentation provided with Laboratory MicroSystems Inc.'s PC/FORTH, version 1, release 26 (this is no longer available). Contact LMI at P.O. Box 10430, Marina del Rey, California 90295.

fig-Forth Cross-Compiler
The documentation for version 1.0 of the Nautilus Systems cross-compiler

Dr. Dobb's Toolbook of Forth, vols. 1 & 2
A fairly wide-ranging treatment of various topics related to Forth. Volume 1 was edited by Marlin Ouverson (and deals with more "nuts and bolts"). Volume 2 was edited by the editors of *Dr. Dobb's Journal*.

Forth Encyclopedia
A detailed description of each fig-Forth word, with source code for all high-level words. By Mitch Derick and Linda Baker, and surprisingly still available from Mountain View Press.

Forth Dimensions, vols. 1–12
A forum for exploring things Forth. Several volumes are out of print.

American National Standard for Information Systems - Programming Language - C
Used as a reference in the articles about translating C to Forth

The C Programming Language
The standard reference for the C programmer. Used primarily for its description of the C language (in the C-to-Forth compiler), and for the heap-management functions found in the last few chapters.

SCO XENIX System V Release 2.3.2, Programmer's Reference Manual and Guides
The documentation provided with the SCO XENIX development system, now in release 4. The description of the Intel relocatable object module format is used as a model for the design of relocatable Forth object modules. The programmer's guides' entries are used for the descriptions of the XENIX/UN*X utilities used in the project.

lex and yacc
A tutorial on the complementary UN*X utilities *lex* and *yacc*. A nutshell handbook by O'Reilly and Associates.

Compilers: Principles, Techniques, and Tools
The second edition of the "Dragon Book." A very dense, basic reference on all aspects of compiler theory, from lexical analysis and parsing to sophisticated optimization strategies. By Alfred Aho, Ravi Sethi, and Jeffrey Ullman (names that figure prominently in the history of UN*X).

80386: A Programming and Design Handbook
Covers all aspects of using an 80386 microprocessor. By Penn and Don Brumm.

386sx Microprocessor, Programmer's Reference Manual
Basically what the title says. Published by Intel.

EGA/VGA: A Programmer's Reference Guide
Used as a reference in the article about character device drivers. By Bradley Kliewer.

AT BIOS KIT: The Comprehensive Guide to Creating an AT BIOS in C
Used as models for the device drivers. By John O. Foster and John P. Choisser, published by Annabooks (12145 Alta Carmel Ct., Suite 250, San Diego, California 92128).

Operating Systems
An overview of OS functions and design issues. By H.M. Deitel.

```
 -- scr # 10
( Basic tools                                          04/09/93 )
BASE @ DECIMAL
: -- ( comment to end of line )
        BLK @ IF IN 64 OVER @ OVER MOD - SWAP +! ELSE
        0 DUP TIB @ ! IN ! THEN ; IMMEDIATE
-- Now the word -- will comment to the end of a line either
-- interpreting from a block file or the terminal
-- Notes: this will not work on a FORTH 83 system
-- This implementation has a misfeature: if the word -- is
-- interpreted at the exact end of a line in a block, it will
-- skip the entire next line!  This is left in because the code
-- to catch the special case would be warty
-->
```

```
 -- scr # 11
( Basic tools                                              04/09/93 )
: AUTOLOAD -- autoload a dependant word from a block
        ( block# -- )
        -FIND IF DROP 2DROP ELSE LOAD THEN ;
-- Used in this idiom
-- -> 27 AUTOLOAD FOO
-- will load screen 27 if the word FOO is not on the default
-- search path.
-- This word probably also won't work unmodified on a FORTH83
-- system.  It is included here only as an example of a kludge
-- that is sometimes used in maintaining libraries of source
-- code in a single block file.  The following variant is
-- PC/FORTH(tm) specific.
: AUTOLOAD_FROM -- autoload a word from a different file
        ( block# -- )
        -FIND IF DROP 2DROP ELSE LOAD-USING THEN ; -->

 -- scr # 12
( Basic tools                                              04/16/93 )
2 CONSTANT CELL : CELL+ 2+ ; : CELL- 2- ; : CELLS 2* ;
: CELL/ 2/ ;
-- These words need to be changed for various implementations
-- If you are using a 32 bit FORTH, the definitions would be
-- 4 CONSTANT CELL : CELL+ CELL + ; : CELL- CELL - ;
-- : CELLS CELL * ; : CELL/ CELL / ;

 -- scr # 13
( ?NUMBER                                                  04/23/93 )
-- Used to convert a number if possible, return a flag
-- stack effect -> ( address -- double \ true | -- false )
HIDDEN DEFINITIONS
: s?num -- do miscellaneous setup for ?number
        ( Address -- 0 \ 0 \ addr' \ -1 \ true \ sign flag )
        0. ROT DUP 1+ C@ ASCII - = DUP >R IF 1+ THEN -1 -1 R> ;
: wt?num -- set up for the while test of ?number
        ( double \ address \ dpl \ flag -- ... )
        ( -- double' \ address' \ true | )
        ( -- double' \ address' \ [true | false] \ false )
        IF DPL ! (NUMBER) DUP C@ BL - IF -1 ELSE -1 0 THEN
        ELSE DROP 0 0 THEN ;
-->

 -- scr # 14
( ?NUMBER                                                  04/23/93 )
: wl?num -- Code for the while loop
        ( double \ addr -- double \ addr \ dpl \ flag )
        DUP C@ ASCII . = 0 SWAP ;
: c?num -- Clean up the output of the loop
        ( double \ addr \ sign \ successf -- double \ true )
        ( | -- false )
        IF SWAP DROP IF DMINUS THEN -1 ELSE 2DROP 2DROP 0 THEN
        ;
```

```
FORTH DEFINITIONS
: ?NUMBER -- The actual word
        ( addr of number -- double \ true | -- false )
        HIDDEN s?num >R BEGIN wt?num WHILE wl?num REPEAT R>
        SWAP c?num ;
FORTH ;S

 -- scr # 15
( Extensions to +!                                      04/23/93 )
: C+! -- Character plus store              ( val \ addr -- )
        SWAP OVER C@ + SWAP C! ;
: |! -- OR store                           ( val \ addr -- )
        SWAP OVER @ OR SWAP ! ;
: C|! -- Character OR store                ( val \ addr -- )
        SWAP OVER C@ OR SWAP C! ;
: &! -- AND store                          ( val \ addr -- )
        SWAP OVER @ AND SWAP ! ;
: C&! -- Character AND store               ( val \ addr -- )
        SWAP OVER C@ AND SWAP C! ;
: ^! -- XOR store                          ( val \ addr -- )
        SWAP OVER @ XOR SWAP ! ;
: C^! -- Character XOR store               ( val \ addr -- )
        SWAP OVER C@ XOR SWAP C! ;
-->

 -- scr # 16
( Extensions to +!                                      04/23/93 )
: -! -- Minus store                        ( val \ addr -- )
        DUP @ ROT - SWAP ! ;
: C-! -- Character minus store             ( val \ addr -- )
        DUP C@ ROT - SWAP ! ;
;S

 -- scr # 17
( CELL/MOD                                              04/23/93 )
-- Faster than `CELL /MOD'
: CELL/MOD -- Divide the value on the top of the stack by the
        -- cell size, leave remainder
        ( val -- rem \ quot )
        DUP 1 AND SWAP 2/ ;
-- On a 32 bit machine, this would be:
-- `: CELL/MOD DUP 3 AND SWAP CELL/ ;
;S
```

ADVERTISERS INDEX

Mixed Integer Arithmetic

Walter J. Rottenkolber
Mariposa, California

Forth follows the assembler paradigm. As a result, data is not typed internally, and the programmer must keep track of what the data on the stack means. In addition, operators are not overloaded, so different functions must be specified for different data types. For example, single and double integers require two separate words for addition, i.e., + and D+.

In *Starting Forth*, Leo Brodie lists a number of mixed arithmetic operators, not part of the Forth-83 Standard, that deal with a combination of double and single integer arithmetic (Table One). Since I occasionally come across source code that uses these functions, I thought it would be handy to have them predefined rather than scurry about patching up something at the last moment.

I use the Laxen & Perry version of Forth-83 since I still run my ancient Z80 system. This Forth includes an eclectic group of mixed and double arithmetic functions, both as part of the Forth-83 Standard and as extensions to it (Table Two). I happily discovered that most of the work had been done already, and only simple extensions were required. Only M*/MOD and M*/ required more extensive coding, and I thank Tim Hendtlass and the Forth Interest Group for the code in UT* and UT/.

All these mixed functions are signed, and no run-time error checking is done. You are obliged to keep both the starting and intermediate values of the signed integers within range (Table Three).

I've noticed, in the programs I use, that almost all division is of positive integers. As a result, I rarely deal with the peculiarities of signed integer division (Table Four). No, nothing is broken. Forth-83 uses floored integer division. When it was implemented in the early 1980s, a great debate ensued between the use of "symmetrical" and "floored" division.

In symmetrical division, the more commonly used, the quotient is rounded toward zero. This leads to a discontinuity in which there are both positive and negative zero quotients, and an inversion of the remainder cycling as the dividend passes from a positive to a negative value. Both the quotient and the remainder track the sign of the dividend.

Floored division rounds toward negative infinity. The sign of the quotient follows a rule similar to that of multiplication, i.e., positive if the signs of the dividend and

Table One.

```
M+        ( d n -- d )
M-        ( d n -- d )
M*        ( d n -- d )
M/MOD     ( d n -- nr nq )
M/        ( d n -- nq )
MMOD      ( d n -- nr )
M*/MOD    ( d n n -- nr nq ) triple intermediate
M*/       ( d n n -- d ) triple intermediate
```

Table Two.

```
D+        ( d d -- d )
D-        ( d d -- d )
UM*       ( u u -- ud )
U*M       ( u u -- ud ) same as UM*
*D        ( n n -- d )
UM/MOD    ( ud u -- ur uq )
MU/MOD    ( ud u -- ur udq )
M/MOD     ( d n -- nr nq )
```

Table Three.

16-bit = n =	-32768 <- 0 -> +32767
32-bit = d =	-2,147,483,648 <- 0 -> +2,147,483,647

Table Four.

dividend	divisor		quotient	remainder
+4000	+27	---	+148	+4
-4000	+27	---	-149	+23
+4000	-27	---	-149	-23
-4000	-27	---	+148	-4

divisor are the same, and negative if they are not the same. The sign of the remainder follows that of the divisor. As a result, zero applies only to a positive quotient, and the remainder cycles through the same values as the dividend passes from a positive to a negative value.

All this may seem to be nit-picking, but since Forth is widely used in embedded systems, a smooth and consistent transition through the zero point was considered valuable in programming such things are plotters and robot arms.

So, a warning. When translating programs from other languages to Forth, you should be aware that not all signed integer divisions are the same, and that computed data must be compared to expected values.

References

Berkey, Robert. "Positive-Divisor Floored Division," *Forth Dimensions* XII/1, May/June 1990, p. 14.

Hendtlass, Tim. "Math—Who Needs It?" *Forth Dimensions* XIV/6, March/April 1993, p. 27 (part one of a two-part series).

Smith, Robert L. "Signed Integer Division," *Dr. Dobb's Journal* #83, September 1983, pp. 86–88.


```
      1
 0 .\  Mixed-length Operators -- M+ etc.
 1
 2 : M+  ( d n -- d-sum )   S>D D+ ;
 3 : M-  ( d n -- d-diff )  S>D D- ;
 4
 5 : M*  ( n n -- d-prod )  *D ;
 6
 7 : M/MOD   ( d n -- n-rem  n-quot ) \ from f83.com
 8     ?DUP  IF  DUP >R  2DUP XOR >R  >R
 9     DABS R@ ABS  UM/MOD  SWAP R> ?NEGATE
10     SWAP R> 0< IF  NEGATE OVER
11     IF  1- R@ ROT - SWAP  THEN THEN
12     R> DROP THEN  ;
13 -->
14
15
```

```
      2
 0 .\  Mixed-length Operators -- M+ etc.
 1
 2 : M/   ( d n -- nquot )  M/MOD NIP ;
 3 : MMOD  ( d n -- nrem )  M/MOD DROP ;
 4
 5 : UT*  ( ud un -- ut ) \ ut= unsigned triple
 6     DUP ROT UM* >R >R UM* 0 R> R> D+ ;
 7
 8 : UT/MOD  ( ut un -- ur udq )
 9     >R R@ UM/MOD SWAP ROT 0 R@ UM/MOD SWAP
10     ROT R> UM/MOD SWAP >R
11     0 2SWAP SWAP D+ R> -ROT ;
12
13 -->
14
15
```

```
    3
.\  Mixed-length Operators -- M+ etc.

: M*/MOD  ( d n n -- nr dq ) \ triple intermediate product
    ?DUP IF DUP >R 3DUP XOR XOR >R >R
    ABS >R DABS R> UT* R@ ABS UT/MOD
    ROT R> ?NEGATE -ROT
    R> 0< IF DNEGATE 2 PICK IF -1. D+
    ROT R@ SWAP - -ROT THEN THEN R> DROP THEN ;

: M*/  ( d n n -- d ) \ triple intermediate product
    ?DUP IF 3DUP XOR XOR >R ABS >R
    ABS -ROT  DABS  ROT UT* R> UT/MOD
    R> 0< IF DNEGATE ROT IF -1. D+
    THEN ELSE ROT DROP THEN THEN ;
```

Towards Natural Language Programming

First steps towards programming in natural language style in the object-oriented Forth natOOF.

Markus Dahm
Aachen, Germany

Forth is often used as the basis of an application language, e.g., for instrumentation, image processing, or device control. Many users of application languages do not want to be experts in programming but want to concentrate on their application. Here Forth offers interactivity, extensibility, the use of arbitray names, and dedicated language constructs.

At the Lehrstuhl für Meßtechnik at the Aachen University of Technology, we have developed image processing hardware, such as high-speed digital filters and transformation processors. In an interdisciplinary group of another project, we strive to design digital workplaces for radiologists. Furthermore, student laboratory work for image processing is carried out. In some of these applications, Forth was or is used. However, the unusual, sometimes quite cryptical, syntax of Forth hinders a quick acceptance of the programming system, especially with non-experts, such as the students, radiologists, and psychologists on our interdisciplinary research team. A step towards usability and acceptance of an application language is to make the programming style resemble the *natural language* of the programmer as much as possible.

This enables non-experts to concentrate on a solution, rather than on a computing syntax.

natOOF offers a first step in this direction to overcome the described restrictions. It was designed to enable non-experts to program their application in a style that is close to their natural language, so they can concentrate on their solution rather than on the syntax of a computer language. This shall be supported in as many mother tongues as possible, since this is the language we all speak best.

The base is the object-oriented Forth *OOF*, which was introduced in issue XIV/1 of *Forth Dimensions* in 1992. Here is a short recap of its essential features:

- It provides Forth's interactivity and extensibility, and the possibility to tailor an application language free of any syntactical constraints.
- It is enhanced with features of object-oriented languages, such as a hierarchical structure of classes and methods, inheritance and multiple polymorphism, plus the security of strong typing.

These main buzzwords of object-oriented languages are explained briefly below. natOOF makes use of these features to enable the programmer to program in a way that closely resembles natural language. An appropriate terminology leads to a different view of source code:

The basic command is a sentence consisting of nouns, prepositions, and at most one verb, where

- *nouns* carry the data,
- *prepositions* make the sentences more readable and select between verbs,
- *verbs* specify the action at a position within the sentence where the natural language would place it, and
- the sentence ends at the end of the line.

Since the verb (operator) may be placed anywhere within the sentence, the notations postfix, infix, or prefix are only special cases of this general concept and can be used (and mixed) in the personal way of programming in natOOF.

For our various applications, we need a programming system that is usable in rapid prototyping for all kinds of users—experts, casual users, and novices. For this purpose, the object-oriented Forth OOF has been developed. Figure One gives a summary of the basic concepts.

What's Object Orientation About?

The fundamental concepts and benefits of object-oriented languages shall be described shortly before diving into natOOF.

Following the paradigm of object-oriented languages in a strict manner, everything that exists in OOF is an *object*. There are integer objects (or, in other words, objects of the class integer) as well as word objects. In fact, words are not quite fashionable in the object-oriented world, since they run on their own, without being linked to an object. The normal executable object is a *method*. A method is invoked when a message is sent to an object, e.g., the message `draw` is sent to a rectangle object. This object knows how to draw itself, and calls the appropriate method that paints a rectangle on the screen. When, on the other hand, the message `draw` is sent to a circle, the circle object calls a different method to

Figure One. Basic principles of natOOF.

The basic command of natOOF is a sentence consisting of nouns, prepositions, and at most one verb, where:

- *nouns* carry the data ..... nouns relate to *objects*
- *prepositions* make the sentences more readable and select between verbs ..... prepositions relate to *keywords*
- *verbs* specify the action at a position within the sentence where the natural language would place it ..... verbs relate to *methods*

paint a circle on the screen.

So we see that each object itself knows best what to make of messages sent to it. There is no need for the programmer to provide various `if`s or `case`s for each new class of objects, they are all built into the run-time system. This feature is called *polymorphism*. When a special class of shaded rectangles shall be defined, we define `ShadedRectangle` as a subclass of the class `Rectangle` and *inherit* all methods defined for ordinary rectangles, e.g., `setCoordinates`. Only the method `draw`, of course, must be redefined such that it paints a shaded rectangle on the screen. So, a hierarchy of classes is built, where each subclass *inherits* and refines (if necessary, redefines) the properties of its parent.

As far as polymorphism is concerned, OOF goes a step further: the method that is to be invoked not only depends on the class of the object the message is sent to, but depends also on the number and classes of additional parameters. This is called *multiple polymorphism* and will be described in more detail later.

All these features enforce a structured approach towards programming. They support a strategy of stepwise refinement, and relieve the programmer of many burdens, such as taking care of special cases. Additionally, OOF's strong typing makes sure that as many errors as possible are detected at compile time, such as sending the message `draw` to an integer. Thus, run-time errors are kept to an absolute minimum, e.g., hardware errors or buffer overflows.

natOOF Comes Natural

Since natOOF is supposed to be close to a natural language, commands are given in the form of a *sentence*. A sentence contains one *verb* at most, and any number of *nouns* and/or *prepositions*. Examples of valid sentences are:

```
turn on Motor
build Histogram of Image
fülle 50 Gramm Pulver in den Behälter
Cursor löschen
```

According to the basic structure of imperative sentences in most European languages, a sentence mostly starts with a verb. But in natOOF, *the verb may be placed at any position in the sentence*. Thus, this concept can be transferred to languages other than English which require or enable a different syntax. This might appear negligible, but in many applications, especially non-expert users feel much more at home when giving commands or writing programs in their mother tongue than in an English-based computer language.

These examples illustrate the basic style ideas of programming in *natOOF*:

- Use verbs to specify the action at a position within the sentence where the natural language would place it.
- Use prepositions to make the sentences more readable.

That sounds pretty simple and is easy to understand. However, this concept is powerful enough to change the style of programming such that programs tend to look like the pseudo-code many people write to describe what the program is supposed to do. But here it is the actual code, both readable and executable; no translation to a computer language is needed.

In the examples above, commands were given in the form of sentences typical for on-line commands in control applications. They made use of predefined verbs. The next sections describe the underlying OOF, show how to define verbs, and give more details of natOOF's features.

natOOF is an Evolution of OOF

There is a direct correspondence between the syntax elements shown above and the structures of the object-oriented Forth OOF.

The *nouns* can be interpreted as *objects* that hold data (e.g., `Buffer`). Forth is an untyped language; OOF, on the other hand, is strongly typed. Every object belongs to a *class* (type). Its class determines what data the object contains and which methods (functions) are defined that work on the object. When an object is called, it pushes itself onto the stack. E.g., `Valve` pushes the object `Valve` onto the stack.

The *verbs* are *methods* that are applied to objects, optionally taking other objects as parameters (e.g., `read`). Methods are not called directly (words in Forth are) but are searched for at run time in the list of methods available for the object that lies on TOS.

The *prepositions* can be interpreted as *objects* of special classes that help to distinguish between similar sentences by means of OOF's multiple polymorphism.

These are the fundamentals that support the basic realization of natOOF. They make an implementation of verbs, prepositions, and nouns in OOF straightforward. The next step is the switch from the classic postfix notation of Forth to the more natural notation of natOOF, which is mostly, but not necessarily, prefix.

From Postfix to Anyfix

The standard notation in Forth, and OOF, is postfix, i.e., first push all operands onto the stack and, finally, apply the operator. Expressed in OOF terms, the method follows a

Figure Two. The lists that OOF's `interpret` searches.

EXECUTE state	COMPILE state
./.	local objects and parameters
methods for object on TOS (if any)	methods for object on TOS (if any)
global objects	global objects
global words	global words
literal (number, char, or string)	literal (number, char, or string)

number of objects, where the last object in the sentence is the object to which the method is applied. In natOOF, the objects shall be treated as usual in OOF: when they are encountered, they are pushed onto the stack. The only problem that remains is the treatment of the method within, rather than at the end, of the sentence. Surprisingly, this requires only a minor change in OOF's `interpret`. First, the normal `interpret` of OOF will be described:

Like Forth, OOF's `interpret` reads words separated by blanks in the input stream. Then the word is searched for in several lists, and is executed or compiled when found, depending on the system's state. Forth searches vocabularies, OOF searches the lists shown in Figure Two.

As explained above, the methods are searched in the list of methods defined for the class of the object on TOS. All methods of the given name are found whose input objects match the objects currently on the stack. Of all methods found, the one with the most input-parameters is chosen. This ensures that

```
12 Degrees Valve open
```

invokes another method than

```
Valve open
```

Both methods open are defined as methods of the class of `Valve` but they execute different pieces of code. This is an example of polymorphism; and since the difference here lies also in the number of parameters, it is also an example for multiple polymorphism.

In order to allow the operator to appear anywhere within a sentence, the treatment of words that were not found or could not be converted to a literal is modified. In Forth and OOF, in this case, an error message is displayed. natOOF's `interpret`, on the other hand, memorizes the first word that could not be found and proceeds interpreting the input stream.

In a regular sentence, the nouns and prepositions are either parameters or local objects of a method or global objects. In either case, they are found and are compiled or pushed onto the stack. But the verb is neither a parameter, a local, nor a global object, and will, therefore, not be found. Its name will be memorized as the potential name of a method. When all operands are given, the sentence is finished and the verb can be evaluated.

Enter the global logical variable `EndOfSentence`. When this variable is true, `interpret` tries to interpret the memorized name in the way described above. When the verb is the name of a valid method, the method is found and compiled or executed. That's all it takes to place the verb somewhere within the sentence.

So, the only thing missing is a way to determine the end of the sentence. One way is to end it explicitly with a period, which sets `EndOfSentence` to `True`. Although this is close to other computer languages where a statement is terminated by a semicolon, it is tedious and a source of errors. So, by default, a sentence is terminated at the end of the line. In this case, <CR> is read from the input stream and a word of that name is executed, which sets `EndOfSentence` to `True`. On the other hand, the characters `...` extend the sentence to the next line, if necessary.

If the verb is not found after the sentence is finished, an error message is displayed and the interpretation is terminated. The same reaction takes place if one of the nouns or prepositions of the sentence is not found.

This strategy is illustrated in Figure Three by the code of natOOF's `interpret`, which is written in natOOF itself to demonstrate its use. The example also shows how to define a method by supplying its name, the input parameters, the output parameters, and local objects. The syntax

```
(( input -- output || local ))
```

is derived from Forth's stack comment

```
( before -- after )
```

Note that all objects are typed, they are defined as objects of a class. The colon definition was generalized to define objects of any class in the format

```
classname : objectname ;
```

For a detailed description, see *Forth Dimensions* May/June 1992 or the *euroFORML 91 Proceedings* for the paper describing the object-oriented Forth OOF.

The assignment statement has now become the verb `:=` which is explained later. The implementation, use, and definition of prepositions and special words, such as `of`, `from`, `into`, `is`, and `the` is explained in the next section.

Prepositions and Other Keywords

Most of natOOF's special features are based on OOF's *multiple polymorphism*. `interpret` makes a lot of decisions, based on the classes of objects that are passed as arguments for a method, which in most other languages the programmer has to do himself in the form of `if...else` or `case` clauses, or with extensive use of flags. natOOF uses *keywords* to differentiate between methods of the same name and arguments. Keywords are just normal objects, if of special classes:

Prepositions, such as `from`, `into`, `of`, `onto`, `under`, etc. are implemented as objects of the classes `_from`, `_into`, `_of`, `_onto`, `_under`, etc., respectively. They can be used, e.g., in

```
readWord into Inword from Instream
```

where they have no function except to make the sentence

Figure Three. natOOF's `interpret`, written in natOOF.

```
Im : interpret      \ define method 'interpret
(( stream : Instream ;  --  \ input -- output
|| string : Inword , Verb ;  ))  \ local  objects

EndOfSentence := FALSE

repeat

  if EndOfSentence
    EndOfSentence := FALSE
    if ( the Verb is empty )
      readWord into Inword from Instream      \ sentence w/o verb
    else
      Inword := Verb                          \ sentence with verb
    endif
  else
    readWord into Inword from Instream          \ within sentence
  endif

  if OOF_COMPILE       \ global logical variable for system state
    compileContents of Inword  \ return True if Inword not found
  else
    executeContents of Inword  \ return True if Inword not found
  endif

  if                                            \ Inword is unknown
    if ( the Verb is empty )
      Verb := Inword       \ keep first unknown as potential verb
    else
      OOFErrorCode := NotFoundError        \ second unknown found
    endif
  endif

until ( ( end of Instream )              or ...
        ( OOFErrorCode <> OOFnoError )            )

end
;               \ end of the definition of the method 'interpret'
```

differentiate between : methods for global objects, and input, output, and local objects of a method. Classically, a global state flag is used which a single : would test. In OOF, for each of these cases, a special : method exists which creates the appropriate kind of object. The methods ((, --, and | | push associated keywords onto the stack which automatically cause `interpret` to find and call the correct :.

This concept follows the tradition of Forth, in that it offers *small dedicated functional blocks* which are called in their specific circumstances, rather than one big general function with a lot of decisions and cases. Thus, special functions and the associated keywords can be added easily. Another advantage is that *it comes for free.* It makes use of the existing features of OOF; no special mechanism, such as lots of global flags or local switches, need be used. It is completely independent of the other significant quality of natOOF, the variable notation, and can therefore be used in normal, postfix-oriented OOF, too.

New keywords can be defined at any time just by defining a new class and an object of this new class. In order to make this process even easier, a class `keyword` is predefined.

The definition

```
keyword : with ;
```

creates both a new class `_with` and the object `with` of this new class, plus a method `with` which is used within the definition of input parameters of a method. As an example, the method `connect` shall be defined that connects a device with a port:

```
Im : connect
     (( device : source ;
        with
        port   : target ;
     --
     ))
```

It can then be used, e.g., in the sentence

more readable. Or they can differentiate methods of the same name but different meanings, as in

```
lay window1 onto window2
```

as opposed to

```
lay window1 under window2
```

Because `onto` and `under` belong to different classes, `interpret` is able to differentiate between the two methods called `lay`. In the first sentence, objects of the classes `window_onto` and `window` lie on the stack, when a method of the name `lay` shall be found. If a method `lay` for objects of the class `window` with exactly these input parameters exists, it is found and executed or compiled, depending on the system's state. In the second sentence, the presence of objects of the classes `window_under` and `window` on the stack cause a different `lay` to be found.

Incidentally, in the same manner OOF is able, e.g., to

```
connect printer with com1
```

Note that, in this example, there are now two objects called `with` in the system. One is a global object of the class `_with` and the other is a method. But `interpret` finds the method `with` only within the definition of input parameters, due to the stack effect of the method `with`. When it is found, `interpret` stops searching and, thus, the global object `with` is not found—because the list of global objects is only searched if no method was found (see the search order stated above). The global object `with` is found in all other circumstances.

Arithmetic and Logical Expressions

For non-experts, especially, arithmetic and logical expressions are easier to read and write when the "normal" infix notation is used. These expressions are special because everyone has learned a way of writing them, whereas few people actually have to handle function calls in everyday life. Any deviation from the schoolday notation is likely to cause irritation and errors.

The arithmetic expression

```
a + b
```

in natOOF consists of two integers a and b, and the arithmetic operator +, as usual. The operator may be placed anywhere in the sentence: `a +` `b`, `a b +`, even `+ a b` are valid sentences and may be used according to the programmer's preferences.

The assignment operator is `:=`, which is used in the form:

```
a := c
```

If a compound expression shall be assigned, it must be evaluated as nested sentences, as in:

```
a := ( b + c )
```

or

```
a := ( ( 3 + b ) * ( c - d ) )
```

Logical expressions and comparisons are evaluated and assigned similarly.

However, in applications, where many calculations must be made, this is still too cumbersome. An entirely different concept would implement := as a parser that interprets the sentence separately. This parser could also implement operator precedence, such as * and / before + and −. This approach turns away from Forth's idea of single smart tools rather than one complex function, but on the other hand promises to provide the easiest, i.e., the usual, syntax for compound expressions.

Special Words `the`, `is`, `it`, `'s` and the Parentheses

Some special words make programming easier and improve readability. They will now be introduced:

- The article `the` can be added anywhere in a sentence. It is a word with no function but to make the sentence look more natural.
- The auxiliary verb `is` was defined for the same reason; it may be used anywhere within a sentence to make it more readable.
- The parentheses `(` and `)` are used to nest sentences, e.g.:

```
read into ( next Buffer ) from Instream
```

or, as shown above in the examples, for nested arithmetic expressions. Here, the object that follows `Buffer` in a list is fetched first; then this buffer is filled from the `Instream`. The implementation is simple: `(` is a slightly modified `interpret` that is terminated by the immediate word `)`.

- The pronoun `it` may be used to refer to the object that was on TOS at the beginning of the sentence. Example:

```
next Buffer .    read into it from Instream .
```

does the same as

```
read into ( next Buffer ) from Instream
```

in the example above, `it` can therefore be used to separate sentences without stackrobatics such as `swap` or `rot`. `it` changes the order of the stack entries by moving, not copying, the referred object to TOS.

- An object in OOF may consist of other objects. E.g., a string object contains the maximum length (`max`), the actual length (`length`), and the text of the string (`text`). These may be accessed only in methods for strings, a feature called *data encapsulation*. The actual length of the string `name` is accessed by

```
name -> length
```

In natOOF, `'s` does exactly the same as `->` but looks more natural:

```
name 's length
```

Extended and Unified Loop Structure

The object-oriented nature of natOOF, especially its multiple polymorphism feature, makes it possible to redefine the loop control structure such that the programmer always uses the same set of control words, regardless of the kind of loop. The idea here is to make looping structures more readable and easier to write. Incidentally, this concept was triggered by Gordon Charleton's talk on this topic at euroFORTH in October 1992 in Southampton, Englad (see also his article "One-Screen Unified Control Structures" in *Forth Dimensions* XIV/6).

The basic frame of a loop is

```
repeat
  ... sentences ...
end
```

which is an endless loop. The `sentences` (remember: natOOF slang for commands or statements) between `repeat` and `end` will be executed until the CPU goes to sleep.

If the loop shall be terminated under one condition or shall continue depending on another condition, this construct is extended to:

```
repeat
  ... sentences ...
until ( logical expression to terminate )
  ... sentences ...
```

```
while ( logical expression to continue )
  ... sentences ...
end
```

In these cases, `repeat` and `end` are words, they can appear without messages being sent to an object. The end of the construct is always marked by `end`. Therefore, in the following loopings, only the `repeat` statement is mentioned. In every loop that will be described, `until` and `while` statements can be used, increasing both the flexibility of the construct and the programmer's control.

The next case is the loop which shall be executed a specific number of times. Hence, the repeat clause looks like:

```
repeat 3 times
```

If the looping index shall be used within the loop, we must define a local integer, e.g., `Index`, and use it as the index:

```
repeat with Index 6 times
```

Now `Index` is the loop count, starting at zero and ending at five. Calling `Index` anywhere within the loop will yield the current loop count. If the boundaries of the loop shall be defined, use

```
repeat with Index from 3 to 8
```

which will cause Index to run from three to eight. The most evolved counted-loop repeat statement is

```
repeat with Index from 3 to 8 step 2
```

introducing the increment of the loop count. This is natOOF's form of Forth's `do...+loop`, which in this case would look like

```
8 3 do ... statements ... 2 +loop
```

where the loop count is returned by the predefined word `i`.

It goes without saying that the start and end indices, as well as the step increment, may be any integer, global objects, results, or local objects, not just literals.

`times`, `with`, `from`, `to`, and `step` are keywords, as described above. These repeat clauses are defined as methods of the class `Integer`. As an example, here is the head of the definition of the last-described repeat clause:

```
Im : repeat
            (( with
               Integer : Count ;
               from
               Integer : Start ;
               to
               Integer : End ;
               step
               Integer : Increment :
            --
            ))
```

In addition to the ordinary loop, it is possible to link a loop to an object that contains a number of elements. One example is an array:

```
repeat for every Index into A
```

initializes `Index` to zero and quits when `Index` reaches the maximum index into `A`, i.e., size of `A - 1`.

Another example is a *collection*. A collection is basically a list of objects. Normally, the elements of a collection are not accessed via an index but sequentially, one after the other. A typical collection contains all objects that are displayed on the screen. If you want to find an object by a coordinate (e.g., because you happened to click the mouse at this coordinate), a collection of display-objects must be traversed, i.e., every display-object must be examined. This is greatly simplified by the basic `repeat` statement for collections. In order to refer to the current display-object, we must define a local shallow display-object, here called `CurrentDO`. Before every looping, `CurrentDO` is updated to refer to the next display-object in the collection, which can thus be manipulated. Figure Four shows how to define the search.

Figure Five shows an overview of all `repeat` structures.

Caveats and Proposed Conventions

There are always gremlins hidden somewhere behind a concept. In natOOF, the major sources of irritation are objects that are forgotten on the stack, e.g., return parameters of methods which are not used. They may lead to misinterpretations of the following code because they can be mistaken as arguments, resulting in either a different method being invoked or no method being found at all. But since the code for methods tends to be short and concise (an inheritance from Forth's well-known and proven strategy of factoring), it is mostly very easy to find the error.

Less important is when verbs and nouns, or prepositions, have the same name. Although this is not very likely for semantic reasons—verbs *do* something, nouns *are* something—the convention below can avoid this error:

- Nouns start with a capital letter and
- verbs and prepositions start with a lower-case letter.

This will also improve readability, since these visual clues help to tell objects and methods apart (see the code of `interpret` above as an illustration).

Future Work

There is a major project coming up where the software will be written in natOOF. This will provide experience and insight into this way of programming, and into its implementation. The integrated environment of OOF will be adapted to natOOF.

natOOF is currently based on the operating system OS9 but will be ported to other hosts, such as Apple Macintosh, PCs, and UNIX workstations. Therefore, the virtual machine, currently written in 68020 assembly language, will be redefined in C with modifications to optimize execution speed.

The portability of natOOF to (European) languages will be evaluated, involving research on the linguistics of natural languages and its significance for natOOF. This shall lead to the definition of more complex sentences. Perhaps the most ambitious goal is to break down a sentence into individual elements which may be used in any order. E.g., if a sentence was defined as

Figure Four. Example of a `repeat-end` structure over a collection.

```
Im : find ((   Point : P ;    in
               Collection of Display-Object : DOs ;
          --   ^ Display-Object : FoundDO ;
\ shallow Display-Object to return to the found Object
          ||   ^ Display-Object : CurrentDO ;
\ local shallow Display-Object to refer to the current DO
          ))

          repeat for every CurrentDO of DOs
             if ( P lies in CurrentDO )
                FoundDO -> CurrentDO
                              \ refer to the found Display-Object
             endif
          end
   ;
```

Figure Five. Overview of `repeat-end` structures.

```
repeat
      ... while ... until ... end
      \ The body of every repeat-end structure can
      \ contain any number of while and until statements.
      \ The following shows only the repeat-clauses:
repeat N times
repeat with Index N times
repeat with Index from Start to End step Inc

repeat for every Index into Array

repeat for every Element of Collection
```

copy N bytes from Source to Destination

it shall be possible to call it by typing
from Source copy N bytes to Destination

without redefining the sentence. This will hopefully lead to fewer necessary references to manuals and to fewer errors. It will be no mean feat to achive this without resorting to megabytes of artificial "intelligence." Stay tuned.

We hope our work will not be misused for military purposes. We will not take part in any military project.

References

"OOF, an Object-Oriented Forth," M. Dahm, *euroFORML '91 Proceedings*, Forth Interest Group.

"Object-Oriented Forth," M. Dahm, *Forth Dimensions*, May/June '92 (XIV/1), Forth Interest Group.

"One-Screen Unified Control Structures," G. Charleton, *Forth Dimensions*, Jan/Feb 1993 (XIV/6), Forth Interest Group.

Markus Dahm earned his Dipl.Ing. (electrical engineering) in 1987 at the University of Technology, Aachen, Germany; and his M.Sc. (computing science) in 1988 at Imperial College, London, U.K. He has been a research assistant since 1989 at Lehrstuhl für Messtechnik in Aachen, working on user interfaces for medical image workstations and natural language programming for non-experts.

More on Numbers

C.H. Ting
San Mateo, California

Sines and Cosines

Sines and cosines of angles are among the most often encountered transcendental functions, useful in drawing circles and many other different applications. They are usually computed using floating-point numbers for accuracy and dynamic range. However, for graphics applications in digital systems, single integers in the range from -32768 to +32767 are sufficient for most purposes. We shall study, using single integers, the computation of sines and cosines.

The value of the sine or cosine of an angle lies between -1.0 and +1.0. We choose to use the integer 10000 in decimal to represent 1.0 in the computation so that the sines and cosines can be represented with enough precision for most applications. Pi is, therefore, 31416, and a 90-degree angle is represented by 15708. Angles are first reduced into the range from -90 to +90 degrees, and then converted to radians in the ranges from -15708 to +15708. From the radians we compute the values of sine and cosine.

The sines and cosines thus computed are accurate to one part in 10,000. See Figure One. (This algorithm was first published by John Bumgarner in *Forth Dimensions* IV/1.) To test the routines, type:

```
90 SIN .        ( 9999 )
45 SIN .        ( 7070 )
30 SIN .        ( 5000 )
 0 SIN .        (    0 )
90 COS .        (    0 )
45 COS .        ( 7071 )
 0 COS .        ( 10000 )
```

Random Numbers

Random numbers are often used in computer simulations and computer games. See Figure Two. This random-number generator was published in Leo Brodie's *Starting Forth*. To test the routine, type:

```
100 CHOOSE .
100 CHOOSE .
100 CHOOSE .
```

and verify that the results are randomly distributed between zero and 99.

Exercise One. Remember Chinese fortune cookies? Write a program which will dispense fortune cookies by selecting a fortune randomly from a fortune database. (You have to build the database first.)

Square Root

There are many ways to take the square root of an integer. The routine shown in Figure Three was first discovered by Wil Baden. Wil used this routine as a programming challenge while attending a FORML Conference in Taiwan (1984).

This algorithm is based on the fact that the square of n+1 is equal to the sum of the square of n plus 2n+1. You start with a 0 on the stack and add to it 1, 3, 5, 7, etc., until the sum is greater than the integer whose root you wish to know. The number when you stop is the square root.

The definite loop structure:

```
( limit index ) DO <repeat-clause> ( inc ) +LOOP
```

is very similar to the DO...LOOP sructure in repeating the repeat clause. The difference is that +LOOP increments the index by the number on top of the data stack. It thus allows the loop index to be incremented or decremented by an arbitrary amount computed at run time. +LOOP terminates the loop when the incremented index is equal to or greater than the limit.

Wil uses +LOOP in an unconventional way. Here the index is used as an accumulator, adding the sequence (2n+1) until the sum exceeds n1. At this point, n2, left on the stack, is the square root of n1.

Exercise Two. Newton's method to find the square root is a very common method used in programming computers. If r1 is an approximation to the square root of n, then a better approximation is:

r2 = (r1 + n/r1)/2

See if you can write a program to find the square root this way.

Exercise Three. The cubes and powers of four of integers can run out of the single-integer range very quickly. It is, therefore, not a bad idea to find the cubic roots or power-of-four roots by raising consecutive integers to the third or fourth power, and compare them to the integer whose root you want. Write a new instruction CubicRoot to find the

cubic root of any positive integer. Try the same for the root of the fourth power as well.

The Greatest Common Divisor

The greatest common divisor (GCD) of two integer numbers is the largest number which can fully divide both the numbers. The most famous method to find the GCD of two numbers m and n was, according to the ancient mathematician Euclid:

If m>n, find GCD(n,m)
If m=0, GCD(m,n)=n
Otherwise, GCD(m,n)=GCD(m,MOD(m,n))

Translating the algorithm to Forth, we have the code shown in Figure Four. This is an excellent example to illustrate the use of an indefinite loop:

```
BEGIN <repeat-clause>
( f ) WHILE
<true-clause> REPEAT
```

which repeats the repeat clause and the true clause if the flag tested by WHILE is true. When that flag is zero, the loop is terminated.

You may want to test the routine by typing:

```
123 456 GCD .
```

Exercise Four. The Least Common Multiple (LCM) of two numbers can be computed by dividing the product of these two numbers by their GCD. Write a new instruction LCM which returns the least common multiple of two integers.

The Fibonacci Sequence

Fibonacci was the pseudonym of the great mathematician Leonardo of Pisa in the Middle Ages. He posed a problem dealing with the offspring generated by a pair of rabbits:

Figure One. Bumgarner's sine and cosine routines.

```
31415 CONSTANT PI
10000 CONSTANT 10K                    ( scaling constant )
VARIABLE XS                           ( square of scaled angle )

: KN ( n1 n2 -- n3, n3=10000-n1*x*x/n2 where x is the angle )
        XS @ SWAP /                   ( x*x/n2 )
        NEGATE 10K */                 ( -n1*x*x/n2 )
        10K +                         ( 10000-n1*x*x/n2 )
        ;

: (SIN) ( x -- sine*10K, x in radian*10K )
        DUP DUP 10K */                ( x*x scaled by 10K )
        XS !                          ( save it in XS )
        10K 72 KN                     ( last term )
        42 KN 20 KN 6 KN              ( terms 3, 2, and 1 )
        10K */                        ( times x )
        ;

: (COS) ( x -- cosine*10K, x in radian*10K )
        DUP 10K */ XS !               ( compute and save x*x )
        10K 56 KN 30 KN 12 KN 2 KN    ( serial expansion )
        ;

: SIN ( degree -- sine*10K )
        PI 180 */                     ( convert to radian )
        (SIN)                         ( compute sine )


: COS ( degree -- cosine*10K )
        PI 180 */
        (COS)
        ;
```

Figure Two. Brodie's random-number generator.

```
VARIABLE RND                          ( seed )
HERE RND !                            ( initialize seed )

: RANDOM ( -- n, a random number within 0 to 65536 )
        RND @ 31421 *                 ( RND*31421 )
        6927 +                        ( RND*31421+6926, mod 65536)
        DUP RND !                     ( refresh the seed )
        ;

: CHOOSE ( n1 -- n2, a random number within 0 to n1 )
        RANDOM UM*                    ( n1*random to a double product)
        SWAP DROP                     ( discard lower part )
        ;                             ( in fact divide by 65536 )
```

A man puts one pair of rabbits in a certain place entirely surrounded by a wall. How many pairs of rabbits can be produced from that pair in a year, if the nature of these rabbits is such that every month each pair bears a new pair which, from the second month on, becomes productive?

The sequence will be: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377...

I propose two equivalent solutions in Figure Five. To test the routines, try:

```
Fib1
10000 Fib2
```

`Fib1` is a routine using explicit instructions to add the two previous Fibonacci numbers to get the next one. It uses the indefinite loop structure:

```
BEGIN
<repeat-clause>
( f ) UNTIL
```

which repeats the repeat clause until the flag becomes true. `UNTIL` terminates the loop when it detects that the top item on the stack is not zero.

`Fib2` uses the implicit addition property of `+LOOP` to compute the next Fibonacci number and compare it to the range limit n. This method is similar to Wil Baden's method for computing the square root of an integer number.

Exercise Five. The binary series is similar to the Fibonacci series, and its members are: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024...

Write a new instruction `BinarySeries` to display this series of numbers.

Figure Three. Baden's square-root routine.

```
: SQRT ( n1 -- n2, n2**2<=n1 )
        0                           ( initial root )
        SWAP 0                      ( set n1 as the limit )
        DO      1 + DUP             ( refresh root )
                2* 1 +              ( 2n+1 )
        +LOOP                       ( add 2n+1 to sum, loop if )
        ;                           ( less than n1, else done )
```

Figure Four. Greatest common divisor.

```
: GCD ( m n -- gcd )
        BEGIN   2DUP >              ( if m>n, exhange m and n )
                IF SWAP THEN
                OVER                ( if m=0, exit loop )
        WHILE   OVER MOD            ( else, replace n by mod[m,n])
        REPEAT                      ( repeat until m=0 )
        SWAP DROP                   ( discard m, which is 0 )
        ;
```

Figure Five. Fibonacci sequence routines.

```
: Fib1 ( -- , print all Fibonacci numbers less than 50000 )
        1 1                         ( two initial Fib numbers )
        BEGIN   OVER U.             ( print the smaller number )
                SWAP OVER +         ( compute next Fib number )
                DUP 50000 U>        ( exit if number too large )
        UNTIL                       ( else repeat )
        2DROP                       ( discard the numbers )
        ;

: Fib2 ( n -- , display all Fibonacci numbers smaller than n )
        1                           ( initial number )
        SWAP 1                      ( set up range )
        DO      DUP U.              ( print current number )
                I                   ( the next Fibonacci number )
                SWAP                ( prepare the next to come )
        +LOOP                       ( add current to index, if the )
                                    ( repeat until sum>n )
        U.                          ( print the last Fib )
        ;
```

Dr. C.H. Ting is a noted Forth authority who has made many significant contributions to Forth and the Forth Interest Group. His tutorial series will continue in succeeding issues of *Forth Dimensions*.

HARVARD SOFTWORKS

NUMBER ONE IN FORTH INNOVATION

(513) 748-0390 P.O. Box 69, Springboro, OH 45066

You already know HS/FORTH gives more speed, power, flexibility and functionality than any other implementation. After all, the majority of the past several years of articles in Forth Dimensions has been on features first developed in HS/FORTH, and many major applications discussed had to be converted to HS/FORTH after their original dialects ran out of steam. Even public domain versions are adopting HS/FORTH like architectures. Isn't it time you tapped into the source as well? Why wait for second hand versions when the original inspiration is more complete and available sooner.

Of course, what you really want to hear about is our **SUMMER SALE!** Thru August 31 only, you can dive into Professional Level for $249. or Production Level for only $299. Also, for each utility purchased, you may select one of equal or lesser cost free.

Naturally, these versions include some recent improvements. Now you can run lots of copies of HS/FORTH from **Microsoft Windows** in text and/or graphics windows with various icons and pif files available for each. Talk about THE tool for hacking Windows! But, face it, what I really like is cranking up the font size so I can still see the characters no matter how late it is. Now that's useful. Of course, you can run bigger, faster programs under DOS just as before. Actually, there is no limit to program size in either case since large programs simply grow into additional segments or even out onto disk.

Good news, we've redone our **DOCUMENTATION!** The big new fonts look really nice and the reorganization, along with some much improved explanations, makes all that functionality so much easier to find. Thanks to excellent documentation, all this awesome power is now relatively easy to learn and to use.

And the Tools & Toys disk includes a complete mouse interface and very flexible menu support in both text and graphics modes. **Update to Revision 5.0,** including new documentation, from all 4.xx revisions is $99. and from older systems, $149. The Tools&Toys update is $15. (shipping $5.US, $10.Canada, $22.foreign)

HS/FORTH runs under MSDOS or PCDOS, or from ROM. Each level includes **all** features of lower ones. Level upgrades: $25. plus price difference between levels. Source code is in ordinary ASCII text files.

HS/FORTH supports megabyte and larger programs & data, and runs as fast as 64k limited Forths, even without automatic optimization -- which accelerates to near assembler language speed. Optimizer, assembler, and tools can load transiently. Resize segments, redefine words, eliminate headers without recompiling. Compile 79 and 83 Standard plus F83 programs.

PERSONAL LEVEL $299.
NEW! Fast direct to video memory text & scaled/clipped/windowed graphics in bit blit windows, mono, cga, ega, vga, all ellipsoids, splines, bezier curves, arcs, turtles; lightning fast pattern drawing even with irregular boundaries; powerful parsing, formatting, file and device I/O; DOS shells; interrupt handlers; call high level Forth from interrupts; single step trace, decompiler; music; compile 40,000 lines per minute, stacks; file search paths; format to strings. software floating point, trig, transcendental, 18 digit integer & scaled integer math; vars: A B * IS C compiles to 4 words, 1..4 dimension var arrays; **automatic optimizer delivers machine code speed.**

PROFESSIONAL LEVEL $399.
hardware floating point - data structures for all data types from simple thru complex 4D var arrays - operations complete thru complex hyperbolics; turnkey, seal; interactive dynamic linker for foreign subroutine libraries; round robin & interrupt driven multitaskers; dynamic string manager; file blocks, sector mapped blocks; x86&7 assemblers.

PRODUCTION LEVEL $499.
Metacompiler: DOS/ROM/direct/indirect; threaded systems start at 200 bytes, Forth cores from 2 kbytes; C data structures & struct+ compiler; MetaGraphics TurboWindow-C library, 200 graphic/window functions, PostScript style line attributes & fonts, viewports.

ONLINE GLOSSARY $ 45.

PROFESSIONAL and PRODUCTION LEVEL EXTENSIONS:

FOOPS+ with multiple inheritance **$ 79.**
TOOLS & TOYS DISK $ 79.
286FORTH or **386FORTH $299.**
16 Megabyte physical address space or gigabyte virtual for programs and data; DOS & BIOS fully and freely available; 32 bit address/operand range with 386.
ROMULUS HS/FORTH from ROM **$ 99.**

Shipping/system: US: $9. Canada: $21. foreign: $49. We accept MC, VISA, & AmEx

An H-P Laserjet/ Deskjet Driver

Charles Curley
Gillette, Wyoming

A driver for Hewlett-Packard PCL Level III in Forth is shown. Forth's CREATE...DOES> facility and numeric text output are illustrated.

Historical Note

The version of Forth used here is fastForth, a 68000 JSR/BSR-threaded Forth described in Curley, "Optimization Considerations," *Forth Dimensions* XIV/5.

The code specific to fastForth is related to I/O vectoring, vocabulary manipulation, and compilation, which will be illustrated below. Implementation on other Forths should be fairly easy. Being figForth in style, variables are initialized at compile time by a value on the stack. F@ is a fast, word-boundary-only, version of @, as F! is a fast, word-aligned version of !. Users of other Forths will have to adjust the code.

ASCII control characters (hex 0 to 1f) will be indicated by name, in brackets: <esc>, <lf>, etc.

Background

Hewlett Packard has offered the PCL printer control language in their laser printers for some years. The H-P Deskjet series offers less expensive inkjet technology with much the same printer control language. Although the PCL language is very flexible, drivers for it can be implemented in Forth readily.

H-P PCL

PCL comes in various levels. The higher the level number, the more recent and, generally, the more extensive the abilities of the printer. Each level is supposed to be a proper superset of the previous level.

The generic PCL command string is begun with an escape character, hex 1b or <esc>. This is followed with a punctuation character which indicates a broad category of the command. Commands which control the printer in general are indicated by the ampersand (&) character. These include the nature of the underscore to be used, the page orientation, margins, etc. Font characteristic commands are indicated by the left parenthesis for the primary character set, and by the right parenthesis for the secondary character set. These commands influence such things as bold, italic, point, and pitch. Graphics and font-download commands are indicated by an asterisk.

The next character is usually a letter, indicating a subcategory of command. This is not always the case, as with the character-set selection commands.

The next character is always one or more decimal numeric characters (0–9), indicating a number. In some cases, such as vertical- or horizontal-movement commands, the value is interpreted literally, and may be signed. In other cases, the numbers are rather arbitrary, such as proportional spacing on (1) or off (0).

A leading minus sign is permitted. This is meaningful in some commands, such as horizontal- and vertical-movement commands. Some commands take the numeric value literally, but a negative value is meaningless, such as pitch or points. In these cases, the result of sending a negative number is unknown, and probably silly.

A leading plus sign is optional where all values are assumed to be positive, such as a point-size command. Where the value may be positive or negative, such as a cursor-movement command, the plus sign is required.

The numeric value is always followed by a letter. If the letter is upper case, the command is ended. If it is lower case, another command in the category follows immediately.

Printer commands that have the same category and subcategory may be chained. The first command is sent with its last character lower case. In subsequent commands, the escape, category, and subcategory characters are omitted. In the last command, the last character is sent upper case. For example, the three commands <esc>(s3T, <esc>(s3B, and <esc>(s2Q may be combined into one command: <esc>(s3b3t2Q. (Note that the periods and commas are punctuation, not part of the commands.) This facility can be used to build custom printer commands.

This suggests that the key to H-P PCL is a sequence of numbers, values of some sort, followed by letters which indicate commands. Hmm... value, command, value, command. Sure sounds familiar to me. Somehow, I don't think we're in BASIC any more, Toto.

Two control character commands select the primary and secondary fonts. This means that two fonts may be defined once in a document, and the user may select between them merely by issuing either <si> or <so>. This is useful if one wishes to switch between two different fonts. For example, one might write an article on H-P Deskjet drivers with the text in proportional CG Times 12 pitch, and the sample code in

```
Scr #  1749
 0 \ load: deskjet printer driver           (  9  4 88 CRC  8:58 )
 1 FILING
 2 FORTH DEFINITIONS  FORGET TASK  \ : task ;
 3 VOCABULARY DESKJET    IMMEDIATE
 4 DESKJET DEFINITIONS
 5
 6
 7
 8 BASE F@ >R  DECIMAL
 9 1 14 +THRU
10
11 FORTH DEFINITIONS                  : TASK ;
12 R> BASE f!  EDITOR FLUSH
13
14
15

Scr #  1750
 0 \ variables: deskjet printer driver      ( 21  2 93 CRC 17:31 )
 1 0 VARIABLE SECOND               8 VARIABLE LNFD
 2 0 VARIABLE BLD   HERE 0 ,         CONSTANT UL
 3 0 VARIABLE ITAL
 4
 5 : FONT?    \  ---  chr | select font according to flag
 6   ASCII ( SECOND F@ IF 1+ THEN ;
 7
 8 : DJSIGN  ROT 0< IF  ASCII -  ELSE  ASCII +  THEN  HOLD ;
 9
10
11
12
13
14
15

Scr #  1751
 0 \ deskjet: sequences                     ( 21  2 93 CRC 17:35 )
 1 : DJSEQ  \ c: c c ... ct ---  | compile seq of ct chars
 2          \ r:    ---   | send to deskjet only
 3   CREATE   [COMPILE] "  HERE C@ 1+ ALLOT
 4   DOES> OUT F@ >R ?PRINT IF  LI EMIT  THEN
 5   COUNT TYPE  R> OUT F! ;
 6
 7 : FSEQ   \ c: c c ... ct ---  | compile seq of ct chars
 8          \ r:    ---   | send to deskjet only
 9   CREATE   [COMPILE] "  HERE C@ 1+ ALLOT
10   DOES> OUT F@ >R ?PRINT IF  LI EMIT  THEN  FONT? EMIT
11   COUNT TYPE  R> OUT F! ;
12
13 \ DJSEQ TEST1 abc"    \ debug dummy
14
15

fastForth on Atari ST              (c) 1985-92 by Charles Curley
   Monday  8/ 3/93 10:12:54
```

fixed-width Courier 10 pitch.

The Code

The start of the code, at screen 1749, establishes a number of system conditions. Line one, for example, ensures that the fastForth filing facility is present. Lines three and four establish the DESKJET vocabulary. Line eight sets the base to decimal, saving the existing base for later restoration. Line nine loads the code. Lines 11 and 12 restore the pre-existing fastForth environment. The phrase EDITOR FLUSH is useful during development, and was left in once the code was completed to allow changes to be developed safely.

On the next screen, we establish five variables. SECOND is used to indicate whether the current font is the primary or secondary font. BLD and UL track whether bold or underlining are active. The way UL is defined takes advantage of the nature of fastForth, in that variables are in-line in the dictionary. This was done to allow an application to save the state of these two variables with a 2@ and restore them later with a 2!. Hardly transportable, but fast. This would be useful for a word processor that allowed headers and footers to be defined, so that they could have their own underline and bold characteristics. Similarly, ITAL is used to track whether italics or upright stroke are in use. There is no variable to track fixed versus proportional spacing, but this could be added if necessary.

LNFD is a holdover from a previous printer driver for the NEC Spinwriter. The Spinwriter allows vertical movement in 48ths of an inch, making a standard six lines per inch movement value eight. Back when this driver was written for the original Deskjet, the vertical movement available was in decipoints (720ths of an inch).

Rather than re-write the word processor, the implementor chose to convert from 48ths of an inch to decipoints within the driver (see >VERTICAL infra). With the Deskjet 500, a 48th of an inch movement command is now available. A line feed would move the print head 8/48ths of an inch to achieve six lines per inch, so LNFD is initialized to eight at compile time.

On lines four and five, we define a word to allow us to follow which font is active, the primary or the secondary. Commands to the two fonts are differentiated by the presence of a left or right parenthesis in the command, so FONT? returns the appropriate character. Note that this works because of a peculiarity of the ASCII character set. Those using EBCDIC or other character sets may have to re-write this word, as well as provide a translation table from their character set to ASCII for the printer's benefit.

There are some commands which insist on an embedded plus sign. The Forth word SIGN is used to insert a minus sign into a string indicating a negative value. However, it inserts nothing into a string if the value is positive. Hence, we provide the word DJSIGN on line eight.

On the next screen, we write two defining words for sending sequences of characters to the current output device. In both cases, we compile a string in-line to be sent to the output device. At run time, the code examines the current output device (with ?PRINT) to see if it is the printer. If it is, we send the escape character; otherwise, we do not send it. This allows us to debug by printing to the screen.

The difference between the two defining words is this: FSEQ is used to send font command sequences. These

```
Scr #  1752
 0 \ deskjet: font designators           (  9  4 88 CRC  9:58 )
 1 \ FSEQ PC-8 10U"
 2 \ FSEQ ROMAN8 8U"
 3
 4 \ FSEQ PC-8DENMARK 11U"
 5 \ FSEQ LATIN1 0N"
 6
 7 \ FSEQ ISO-UK 1E"
 8 \ FSEQ ASCII06 0U"
 9
10 \ FSEQ ASCII14 0K"
11
12
13
14
15

Scr #  1753
 0 \ deskjet: font designators           ( 21  2 93 CRC 17:35 )
 1 FSEQ COURIER s3T"
 2 FSEQ CGTIMES s4101T"
 3
 4 : PRIMARY    SECOND OFF  \ select primary font.
 5   ?PRINT IF CTL O ( shift  in ) EMIT THEN ;
 6
 7 : SECONDARY  SECOND ON   \ select secondary font.
 8   ?PRINT IF CTL N ( shift out ) EMIT THEN ;
 9
10
11
12
13
14
15

Scr #  1754
 0 \ deskjet: subscripts italics bold    ( 23  2 93 CRC 10:44 )
 1 FSEQ SUBSCRIPT s-1U"
 2 FSEQ -SCRIPT s0U"
 3 FSEQ SUPERSCRIPT s1U"
 4
 5 FSEQ PROPORTIONAL s1P"
 6 FSEQ FIXED s0P"
 7
 8 FSEQ (ITALIC) s1S"              : ITALIC    (ITALIC) ITAL ON  ;
 9 FSEQ (-ITALIC) s0S"             : -ITALIC  (-ITALIC) ITAL OFF ;
10
11 FSEQ (BOLD) s3B"                : BOLD       (BOLD) BLD ON  ;
12 FSEQ (-BOLD) s0B"               : -BOLD     (-BOLD) BLD OFF ;
13
14
15

   fastForth on Atari ST              (c) 1985-92 by Charles Curley
   Monday  8/ 3/93 10:12:56
```

```
Scr #  1755
 0 \ deskjet: character height              ( 23  2 93 CRC 11:39 )
 1 : BUILDCMD      \ n chr1 chr2 ch3 ---  string typed
 2   OUT F@ >R  ?PRINT IF LI EMIT THEN  BASE F@ >R DECIMAL
 3   >R >R >R  DUP S->D DABS
 4   <#  R> HOLD  #S  DJSIGN  R> HOLD  R> HOLD #>
 5   TYPE  R> BASE F!  R> OUT F! ;
 6
 7 : POINTS   \ pts ---  | set font height in points, 72nd of inch
 8   ASCII V  ASCII s  FONT?  BUILDCMD ;
 9
10 : POINTER  CREATE  W,  DOES> W@ POINTS ;
11
12 24 POINTER 24POINT
13 12 POINTER 12POINT
14  6 POINTER  6POINT
15

Scr #  1756
 0 \ deskjet: character pitch               (  9  4 88 CRC 10:48 )
 1 : PITCH    \ pts ---  | set font pitch, in chrs/inch
 2   ASCII H  ASCII s  FONT?  BUILDCMD ;
 3
 4 : PITCHER  CREATE  W,  DOES> W@ PITCH  ;
 5
 6 DECIMAL
 7  5 PITCHER  5PITCH
 8 10 PITCHER 10PITCH
 9 16 PITCHER 16PITCH
10 20 PITCHER 20PITCH
11
12
13
14
15

Scr #  1757
 0 \ deskjet: underscores,           (  9  4 88 CRC 13:16 )
 1 DJSEQ (1FIXUL) &d1D"
 2 DJSEQ (2FIXUL) &d2D"
 3 DJSEQ (1FLOUL) &d3D"
 4 DJSEQ (2FLOUL) &d4D"
 5 DJSEQ    (-UL) &d@"
 6
 7 ' (1FIXUL) VARIABLE 'UNDER
 8 : LINER  CREATE  [COMPILE] ' , DOES> F@ 'UNDER F! ;
 9 LINER 1FIXUL (1FIXUL)
10 LINER 2FIXUL (2FIXUL)
11 LINER 1FLOUL (1FLOUL)
12 LINER 2FLOUL (2FLOUL)
13 : -UL        (-UL)  UL OFF ;
14
15 : UNDERLINE  UL ON  'UNDER @EXECUTE STOP

fastForth on Atari ST              (c) 1985-92 by Charles Curley
   Monday  8/ 3/93 10:13:00
```

words send commands that control either the primary or the secondary font. All other sequences should be defined with DJSEQ, which does not send any special characters of its own.

In either case, the string laid down at compile time is then sent to the current output device.

These two defining words are useful for defining commands where the numeric value is fixed, such as those which set bold or normal weight, italic or upright style, or which select character sets.

These could also have been used to define fixed pitch and point sizes, but a different route was selected for that (see screens 1755 and 1756).

On screen 1752, there are defined several character-set selectors. As I don't use these, I see no reason to have them occupy dictionary space, and so have them commented out.

I do, however, use both Courier and CG Times typefaces, so these are compiled on screen 1753. Others may be defined ad libitum.

Below these, on lines four through eight, are two words to select between the primary and secondary fonts. These two words maintain the variable SECOND, which indicates the font currently in use.

On the next screen, several words to control font characteristics are defined. These control super- and subscripting, bolding, proportional or fixed spacing, and italicizing. Where variables are defined, these words maintain the relevant variables.

Thus far, we have dealt only in commands in which the numeric portion is fixed. We now look at commands where the numeric field may be influenced by external factors. An example of this is cursor movement. A word processor may deal in multiple

columns, and may wish to start each column at a precise, fixed, horizontal position. The word processor will have to calculate this value and feed it to the driver. It is then up to the driver to embed the value into the appropriate command. On screen 1755, we begin to deal with commands of this nature.

Forth Numeric Output

At this point, a brief digression into Forth's numeric output system is in order.

Forth stores all numbers internally as binary data. Binary Coded Decimal (BCD) is not used. Integer values may, thus, range the full storage capability of the data word (or cell, in dpANS Forthese) on a given implementation.

The typical Forth system for numeric output converts binary values to text by a process of dividing by the contents of the variable BASE. Thus, the output string is written into memory a character at a time, from right to left. Typically, a double-precision value is placed on the stack. It may be tested for sign, if signed output is desired.

Conversion is commenced with the word <# ("begin sharp"), which initializes the relevant variables. One digit may be converted and added to the string with the word # ("sharp"). The value under conversion may be rendered into text until it is exhausted by the word #S ("sharps"). Single characters may be added to the string with the word HOLD. SIGN consumes the single-precision value under the double-precision value being converted to insert a minus sign if needed into the string under construction. Conversion is ended with the word #> ("end sharp"), which consumes a double-precision value from the stack (presumably, the detritus of conver-

```
Scr #  1758
 0 \ deskjet: paper sizes vmi                ( 23  2 93 CRC 11:37 )
 1 DJSEQ 8.5X11 &12A"
 2 DJSEQ 8.5X14 &13A"
 3 DJSEQ A4 &126A"
 4 DJSEQ ENVEL &181A"
 5 DJSEQ DEFAULTPAPER &10A"
 6
 7 : >VERTICAL  \ n ---  | move printhead 48th of in verticaly
 8   15 *   ( 48th -> 720th of in )
 9   ASCII V  ASCII a  ASCII &  BUILDCMD ;
10
11
12
13
14
15

Scr #  1759
 0 \ deskjet: lpi paper feed draft/lq        ( 21  2 93 CRC 17:38 )
 1 DJSEQ (8LPI) &18D"
 2 DJSEQ (6LPI) &16D"
 3
 4 : 8LPI  (8LPI)  6 LNFD F! ;
 5 : 6LPI  (6LPI)  8 LNFD F! ;
 6
 7 DJSEQ EJECT &10H"
 8 DJSEQ PAPER &11H"
 9 DJSEQ ENVELOPES &13H"
10
11 FSEQ DRAFTQ s0Q"
12 FSEQ LETTERQ s2Q"
13
14
15

Scr #  1760
 0 \ deskjet: topm tlen perf -perf           ( 23  4 88 CRC  9:00 )
 1 : TOPM     \ lns ---  | set top margin in lines
 2   ASCII E  ASCII l  ASCII &  BUILDCMD ;
 3
 4 : TLEN     \ lns ---  | set text length in lines
 5   ASCII F  ASCII l  ASCII &  BUILDCMD ;
 6
 7 DJSEQ +PERF &11L"  \  enable perforation skip
 8 DJSEQ -PERF &10L"  \  disable perforation skip
 9
10 : WPSETUP   +PERF 3 TOPM  63 TLEN ;
11
12
13
14
15

   fastForth on Atari ST              (c) 1985-92 by Charles Curley
   Monday  8/ 3/93 10:13:02
```

```
Scr #  1761
 0 \ deskjet: margin setup                    ( 20  5 88 CRC 10:28 )
 1 : LMARGIN  \ col ---  | set left margin in columns
 2   ASCII L  ASCII a  ASCII &  BUILDCMD ;
 3
 4 : RMARGIN  \ col ---  | set right margin in columns
 5   ASCII M  ASCII a  ASCII &  BUILDCMD ;
 6
 7 DJSEQ -MARGINS 9"  \  disable margin setup
 8
 9 DJSEQ L->R &k0W"   \  print left to right only
10 DJSEQ BI &k1W"     \  swing both ways
11
12
13
14
15

Scr #  1762
 0 \ deskjet: output device                   ( 29  7 88 CRC 17:50 )
 1 8 VARIABLE VERT  \ single space it
 2 : VERTICAL  CREATE   W,  DOES>  W@ VERT F! ;
 3
 4 : DJCR  VERT F@ DUP >VERTICAL  LNCTR +!
 5   PRINTER  LNCTR F@ LN/PG F@ >  IF  PPAGE
 6   ELSE CTL M EMIT  LMRGN  THEN ;
 7
 8 DECIMAL FORTH DEFINITIONS  DESKJET
 9  8 VERTICAL SINGLESPACE          16 VERTICAL DOUBLESPACE
10 12 VERTICAL 1.5SPACE             20 VERTICAL 2.5SPACE
11 : [PRINT]   LNCTR  OFF  OUTPUT> PRINTER (PNT)   (TYPE)  2DROP
12   PPAGE   NLIST   DESKJET DJCR  STOP
13 : PRINT  EDITOR FLUSH  [PRINT]  DESKJET +PERF ;
14 : VTAB   DESKJET ?PRINT IF  VERT F@ * DUP >VERTICAL LNCTR +!
15   ELSE  VTAB  THEN ;
```

sion) and leaves an address and count ready for TYPE or other text output words. The location where the string is built is system specific, not re-entrant. The system may guarantee the buffer to be valid only between <# and #> or shortly thereafter. This possibility encourages the programmer to use the string as quickly as possible.

Conversion to any reasonable number base, and some unreasonable ones, may be done by changing the contents of the variable BASE. This may be done during or before text conversion.

To illustrate how this all works, we will examine the fastForth word D.R, identical to the fig-Forth version. It is used to place the signed output of a double-precision value, right justified, in a field of given size:

```
: D.R
\ d fld ---   | type signed d in field of
\               size fld
  >R SWAP  OVER  DABS  <#  #S  SIGN #>
  R> OVER  -  SPACES  TYPE ;
```

>R places the field size onto the return stack for later use. The phrase SWAP OVER gives us a sign flag on the stack, below the double-precision value for conversion. This sign flag will later be consumed by SIGN. We then convert our value to an unsigned value, ready to be converted. Conversion is started with <#. The entire value is converted to a string using #S, leaving a double-precision value of 0 on the stack. SIGN does its thing, optionally adding a minus sign to the string, and eating the signed value left earlier. We end conversion with #>, which eats the double-precision 0, and leave the address and count of the string on the stack. We now recover the field size from the return stack, subtract the length of the string from it, and print the appropriate number of spaces. Finally, we TYPE the string out.

Numeric Commands

With that understood, we are prepared to build the code needed to embed numbers into commands. This is done with the word BUILDCMD, on screen 1755. This word takes as its arguments three ASCII characters and a single-precision number. These will be built into a string, and the string TYPEd to the current output device.

On line two, we save two variables to the return stack. OUT is used to count text output in a line. These commands should be transparent to text output, so we fake it by preserving its contents, for restoration on line five. Similarly, we must be in decimal, regardless of the current base. On the same line, if the current output device is the printer, we emit an escape to indicate the start of a command.

On line three, we save the three characters to the return stack. This is to allow the code to manipulate the numeric argument, which we do immediately. The DUP places a copy of the number on the stack. The lower copy will be used by DJSIGN at the end of numeric conversion. The upper copy is sign-extended to a double-precision value and converted to an absolute value, ready for numeric conversion.

Line four begins string construction. The phrase R> HOLD adds character one to the string under construction. #S then converts the number value to a string, leaving a double-precision 0 on the stack, to be removed by #> later. DJSIGN is executed in order to force either a plus or a minus sign into the string. Twice more we execute the phrase R> HOLD to add the remaining two characters to the string. #> consumes the

double-precision 0 and leaves the address and count of the string, ready for TYPE on line five. We then restore our variables from the return stack, and away we go.

Because string construction is from right to left, the end-of-command character is the right-hand one in the stack diagram, char3, and the category character is on the left, char1. This makes for very readable code, as illustrated by POINTS on line seven of the same screen. The characters are simply in the reverse order of their positions in the output string. In the case of the points command, the syntax is <esc>(s###V for the primary font.

The defining word POINTER lets the user define point sizes he might wish to use routinely. Alternatively, he can use POINTS to command oddball point sizes, subject to the printer's limitations.

On screen 1757, we make arrangements to allow a consistent underline style throughout a document. The printer syntax is to have to set the underline type each time the user wants to add underlining. This requires that the user remember which flavor of underlining he wants. Instead, we remember it for him with the variable 'UNDER. Daughter words of the defining word LINER allow the user to specify his preference for underline style, and the words UNDERLINE and -UL turn underlining on and off, respectively.

```
Scr #  1763
 0 \ deskjet: dj                                    (  9  4 88 CRC 11:35 )
 1 FORTH DEFINITIONS
 2 : DJ  BL WORD HERE COUNT UPPER
 3   HERE [ ' DESKJET C>P ] LITERAL F@  (FIND)  0= 0 ?ERROR
 4   STATE F@ <  IF  <COMP>  ELSE  EXECUTE  THEN  ;
 5 IMMEDIATE       ;S
 6    Allows instant access to a deskjet word.
 7    It finds the word in the deskjet vocabulary, and either
 8 compiles it or executes it, according to its immediate
 9 bit and the contents of  STATE .
10    N.B. Will also find words in the  FORTH  vocabulary.
11    This is a specific version of Brad Rodriguez'  FROM , which
12 is used to get a word from a designated vocabulary, e.g.
13             FROM EDITOR -TEXT
14    Instead, use it as follows
15   ...    DJ BOLD     ...   ...   DJ -BOLD  ...


Scr # 1780
 0 \ deskjet c^2                                   ( 20 10 88 CRC 15:55 )
 1 DESKJET  FORTH DEFINITIONS    FORGET TASK
 2 ( : TASK ; )   BASE @ DECIMAL >R
 3
 4 : C^2 ?PRINT ?OUTFILE OR        \ print c^2 logo on deskjet
 5   IF  DESKJET  5PITCH  ASCII C EMIT  10PITCH
 6       SUPERSCRIPT 6POINT ASCII 2 EMIT 12POINT -SCRIPT
 7   ELSE  ." C^2" THEN ;
 8
 9 : TASK ;
10 R> BASE ! EDITOR FLUSH
11
12
13
14
15
```

The next several screens are straight definitions of various PCL commands.

On screen 1762, we handle the need to track our position on the paper during printing. This capability allows applications to stop printing a document, print footers, go to the next page, print headers, and then resume printing the document. It also allows an application to force text onto the next page in order to preserve a section intact. It also allows proper handling of footnotes.

The key to a Deskjet carriage return is to move the printer vertically with the word >VERTICAL followed by paging, if necessary, and then moving the carriage to the left margin. Paging and moving to the left margin are handled in a raw printer driver in the PRINTER vocabulary, and are not shown here.

We then establish the defining word VERTICAL and use that to make several spacing options available in the FORTH vocabulary.

Almost the last thing to do is to establish a method of hooking the driver into fastForth's I/O system. In fastForth, the words EMIT, TYPE, GOTOXY, PAGE, LIST, and CR are all vectored through the user area. The word OUTPUT> allows the programmer to construct a list of words to fill those vectors. This is done in the word [PRINT] on line 11. The run-time code associated with OUTPUT> stuffs the execution vectors and then returns execution to the calling word. This means that if we want to condition the printer after setting the vectors, we must do that in a calling word. This is done in PRINT on line 13. Other printing conditioning could be added after the call to +PERF.

Finally on this screen, we define a word for vertical tabbing. When the printer is the active output device, we simply command vertical movement; otherwise, we use the native vertical tabbing word.

It would be convenient to be able to reach into the DESKJET vocabulary from time to time without playing

games with vocabularies. One could implement Brad Rodriguez's word FROM, which reaches into a given vocabulary for a one-time access. (e.g., FROM EDITOR -TEXT) Instead, we provide a one-shot word for the DESKJET vocabulary alone.

This access word is defined on screen 1763, and has some fastForth-specific features. On line two, it grabs the next word name to be found, and conditions it at HERE. The code on line three searches for it in the DESKJET vocabulary. As fastForth has fig-Forth style vocabularies, this will also search the FORTH vocabulary. The syntax on line three is specific to fastForth and will have to be modified for other Forths.

If the word isn't found, the code errors out with ?ERROR. On line four, we examine the compile state, and compile or execute, as appropriate. The word <COMP> is specific to fastForth and handles compilation. Indirect-threaded Forths should use the word , (comma) instead.

Sample Application

To illustrate how this code can be used, a simple application will be shown. As my initials are C.C., I use the figure of a large C with a superscript 2, indicating C squared, for my return address on envelopes and elsewhere. This monogram is easily coded, as seen on screen 1780.

If the current output device is the printer or an output file, we implement the fancy logo. Otherwise, on line seven, the proof-text string C^2 is printed.

We enter the DESKJET vocabulary to have access to its drivers. We set the printer to a pitch of five (five characters per inch). We then print the upper-case C. We then go to ten characters per inch. We then set up for a six-point (6/72nds of an inch high) superscript. The superscript 2 is then printed. We then return to our normal 12-point text. Then we restore printing from the superscript line to the base line.

Further Enhancements

One thing not done here is to track more variables. The driver could track, and make available to the application, the current horizontal position (in, say, decipoints), the current pitch and point size, and other variables. One of the problems with the application word C^2 is that it leaves the printer in 12 point and 10 pitch. Point and pitch could have been saved going in, and then restored at the end.

Horizontal movement is dealt with crudely: all type is assumed to be monotype. This is fine for the Courier typeface, but doesn't work in CG Times, a proportional typeface. A proportional typeface might require some look-ahead to determine where to break lines. To get right justification, we would have to add a variable to hold padding values for characters and word gaps.

Another thing not implemented here is cursor movement. Vertical movement is present in 48ths of an inch increments. A re-write to give decipoint and character cursor movement would make multi-column printing and precise placement of graphics from a word processor very easy.

Availability

The code shown here is released to the public domain. Enjoy it in good health, and raise a toast to those who contribute to the public domain from time to time.

Charles Curley, a long-time Forth nuclear guru, lives in Wyoming.

Fast FORTHward

The Impact of Open Systems

Mike Elola
San Jose, California

To better understand why open systems are of such key importance to the future of computing, you may wish to read *UNIX, POSIX and Open Systems* (Addison-Wesley). Here is how John S. Quarterman and Susanne Wilhelm describe the market's motivation to acquire open systems:

"Users want to move programs among machines, and employers want to move users among machines. Vendors want to sell applications for a variety of platforms, and platforms for many applications. Everybody wants applications that can communicate and systems that can exchange data and information..."

Programmers enjoy the benefits of an open development environment whenever they use debuggers, compilers, editors, and libraries from a variety of vendors in tandem on one project. But the path that interoperability has traveled has been a haphazard one. I'll examine some of the changes in the course of computing that helped guide the market and software developers towards open systems.

Some thirty years ago, higher-level programming languages were thought to hold great potential as the foundation upon which applications could take form. Algol, BASIC, and Fortran were among the programming languages thought to be able to displace non-portable assembly language programming. The application programmer would be trained to use a high-level programming language. That training and an acquired understanding of an application domain should lead to a successful application.

Now is the time to use Forth as a labor-saving, service-generating engine that fits into open systems models.

About twenty years ago, timesharing multi-user operating systems became commonplace for mainframes and minicomputers. These operating systems absorbed many of the services once provided by early programming languages, job control languages, and computer operators.

As integrated circuits became more dense, RAM memory devices and microprocessors reshaped the industry. As part of a memory-resident kernel of services, an application service routine could rapidly gain control of the computer. As processor performance and memory increased, the overhead of a co-resident operating system eventually became a moot objection.

Presently, single-user PC operating systems are heavily laden with services too, often making PCs very memory-hungry. Graphical user interfaces are fueling this growth, as well as other services with end-user appeal, such as networking.

Evolving tools and software architectures are making it easier for new layers of software to be introduced. Libraries have long provided a tool for isolating software into layers. Now, dynamic linking adds yet another tool for layering software.

(I recently dropped a system extension into my Macintosh's system folder, which expanded my system's services to include support for DOS file systems on 3.5" disks. This was achieved so easily due to the Mac's layering of software. Without changing any of my applications, I can now issue commands from them to read or write a DOS-formatted floppy.)

In our present situation, it is hardly likely that a trend towards downsized operating systems will develop. Today, the construction of applications takes place upon a substrate of operating system services, GUI services, and a number of other specialized API services.

A key to the success of system software involves satisfying the desire of computer users for consistent services. Users don't want to learn six different ways to establish a network connection or three different ways to print within three different applications.

These APIs have continuously eroded the role of the programming language down to a lesser role: that of enabling communications and coordination of the more important services provided outside the language itself.

Microsoft has become one of the most profitable and most admired companies within the U.S. economy. DOS and Windows are the royal jewels of this industry because of the ever-increasing value of the markets they created. To ensure more open access to the royal jewels of the future, open specifications for system services are anxiously awaited.

Microsoft wants its applications to be among the first to reach any newly opened markets.

Open systems will inevitably grab a large share of the operating systems market. Plug-and-play layers of software will delight users through the system extensibility they afford.

Besides providing the consistency users want, API services are labor savers. For example, graphical applications are no longer directly responsible for drawing window frames, or for closing, resizing, or relocating windows. When you consider that half of an application's code supports the human interface, a GUI API represents a very large reduction in effort for application programmers. Cross-platform GUI APIs will soon lead to much more code reuse and labor savings.

Labor-saving concerns may override programmer concerns over the choice of language. Fourth generation database languages can also lead to labor savings, but they tend to be monolithic systems. The granularity of software layers (APIs) has more potential than 4GLs to open up new markets. This is because of the broad range of applications areas that layers of software can address.

A large volume purchaser of computers and software wants to avoid specifying a particular vendor's make and model. Why discourage competition? So the federal government specifies interfaces describing how unbundled software pieces should fit together, then asks several vendors to bid on the development of each piece. To help establish software components that use the same interfaces and protocols, vendors are now proactively seeking open specifications.

"Interface specifications are the key to open systems.... Open specifications are clearly needed for network protocols, so that various vendors can produce products that will interoperate. They are also important for interfaces to those protocols, so application writers can write applications that will be portable to vendors' systems. Open specifications for interfaces to basic operating system facilities are just as important for portability. User interfaces also require open specifications, if users are to be able to use more than one vendor's products. Open specifications are the key to distributed computing."

The authors also say that the best open specifications are written by third parties. By third parties, they mean standardization committees. Under the guidance of ISO, ANSI, or IEEE rules, such committees use a formal and open process to arrive at open specifications. An open process allows all interested parties to provide input and allows for updates to standards to track innovations in hardware and software.

What does this all mean to the average Forth programmer? It means that now is the time to use Forth as a labor-saving, service-generating engine that fits into open systems models. To meet industry standards, it will mean adding conventional library tools to Forth.

If Forth is proven suitable for implementing many services, we can ensure that Forth has a place at the ground level, if not also at the application level of future computing and development environments. Forth has to become a client of standard system services as well as retain the ability to be a generator of such system services to enter the main stream of software development.

It would be a compelling demonstration of Forth's breadth of utility to use it in a service-generating role. For example, Forth could efficiently and effectively generate the file manipulation and file I/O services that the POSIX standards deem necessary, amounting to forty or more routines. (POSIX refers to Portable Operating System Inter-

(Continues on page 42.)

Product Watch

MARCH 1993

J. Perham announced the availability of *EuroForth 1992 Conference Proceedings* (£17.50 plus postage and packing) from MicroProcessor Engineering Ltd.

MicroProcessor Engineering announced PINC PowerForth, a complete Forth system written in C. To compile the Forth system from C source, use any K&R or proposed ANS C compiler. It is supplied with support for the SunOS CGI and PIXWIN interfaces. The cost is £900.

MicroProcessor Engineering also announced MPE Modular Forth version 3.6 for real-time applications on PCs, including embedded PCs (80x86, Forth-83). It features a window manager, interrupt-driven serial drivers, file browser, and on-line help system. To break the limit of ten 64K maximum-size modules, its dynamic linker can be used as an overlay manager. Source code is provided for the windows and serial driver packages, for the menuing package, for the on-line help system, and for the graphics package. Finally, the standard DOS linker can be used to link to any package that creates a DOS standard .OBJ file, to help support mixed language programming. The price is £625.

FORTH, Inc. announced an add-on GUI option for its polyForth for 80386/486 systems with VGA. The option is an OSF/Motif-style GUI toolkit. It offers low-level support in the form of "widgets," as well as high-level support for dialog boxes and menus. With the toolkit comes a new version of a full-screen Forth editor. Source code for the toolkit is included.

Companies Mentioned

FORTH, Inc.
111 N. Sepulveda Boulevard
Manhattan Beach, California 90266-6847
Fax: 213-372-8493
Phone: 800-55-FORTH

MicroProcessor Engineering, Ltd.
133 Hill Lane
Shirley, Southampton SO1 5AF
England
Telex: 474695 FORMAN G
Fax: 0703 339391
Phone: 0703 631441

One-Screen Virtual Dictionary

Gordon Charlton
Hayes, Middlesex, U.K.

I do not feel comfortable with files. Forth is a static language, at least with reference to its memory allocation routines, and from this it derives much of its simplicity. Files, on the other hand, use disk space in a dynamic fashion. They are, consequently, rather more complex, both in the amount of code required to implement them and in the complexity of code in programs that use them.

Blocks are simple, in comparison, being merely 1K windows into a larger virtual memory. Although this probably resides on disk, it need not. All that is required of BLOCK is that it will accept a number and return a 1K chunk of data that corresponds to that particular number. What BLOCK does not provide is any means of organizing this secondary memory system.

I have seen some attempts to structure block space using dynamic allocation schemes, but on the whole they tend to be overly complex and, in trying to satisfy every possible need, end up not meeting any. If we chose a static allocation system, however, we already have a good model to work from: the Forth dictionary.

It is often forgotten that blocks represent a window into virtual memory...

The program presented here allows just that, and it is simple. In fact, it is simple enough to fit into one screen, although for reasons not entirely unconnected with readability, it is presented on three.

The only thing about BLOCK that will cause us problems is its insistence in slicing memory into monotonously similar 1K portions. This we can compensate for, and end up with a system which is, for me, perfectly adequate for most situations.

Addressing Virtual Memory

Rather than dealing with 1K chunks, we will treat virtual memory as far as possible as contiguous, and addressable within the same range as the host Forth's real memory. This means that for a 16-bit Forth we can address 64K of virtual memory, and for a 32-bit Forth 4 megs. I will refer to the 16-bit case only; if your Forth has a different cell width (as mine does), please change the numbers accordingly.

If one wishes to use more than 64K, we will use 64K pages. 64K chunks will cause us fewer problems than 1K chunks. In fact, we will turn them, as far as we can, to our advantage by allowing not just 64K chunks, but any multiple of 1K, up to 64. These we will refer to by name, as well as by a unique identifier, one cell long, to allow them to be manipulated under program control. This makes them very much like vocabularies.

The first word I define is the constant B/BLK (bytes per block), although in fact it remains a "magic number," even though named, as I make much use of the fact that it is a power of two and consequently allows certain bit manipulations which simplify the math required. I make no excuse for this, as blocks are infrequently, if not never, any other length—and even when they are, they tend to be some power of two bytes in length.

At any time, one page will be addressable, and its identifier available in the variable ACTIVE. A page is declared using V.PAGE. V.PAGE takes a block number as an argument. This is the base block for the page. At instantiation, the dictionary pointer for that particular page of virtual memory is set to zero. A page is made active by mentioning its name, or by storing its identifier into the variable ACTIVE.

The base block for the currently active page is available at the address returned by V.BASE. This, in practice, turns out not to be terribly useful. It could be used in calculating the last block a page occupies, but I usually have a fair idea how large my data structures are anyway, so have not implemented that function. It would be quite simple, if required. V.BASE is included because I habitually try to name fields in composite data structures rather than deal directly with offsets. More usefully, V.DP gives us access to the dictionary pointer for the active page, although generally this should be addressed by V.HERE and V.ALLOT. These are analogous to HERE and ALLOT, hence the names.

Aligning Within Blocks

Addressing a datum that lies completely within one block is easier than addressing one that straddles one or more block boundaries. This probably goes without saying. Therefore, whatever we can do to encourage entries into the virtual dictionary to fit neatly within blocks should be done. V.ALIGN and V.ALIGNED are used for this. V.ALIGNED takes an offset in bytes (again I make an assumption about your Forth, that its address units are bytes) and increases it to the next power of two, unless it is a power of two itself, in which case it does not alter it. (Do not pass it an argument of zero, it does not like it.) Objects whose lengths have been aligned pack neatly into 1K blocks with no gaps. Equally, they do not cross block boundaries, provided that they start at an appropriate position.

To illustrate, assume I have an array of 200 items, each 12 bytes long. If I align them, I will be dealing with an array of items 16 bytes long, four of which will not be used. For the cost of this wasted space, I gain the advantage of them fitting neatly, 64 to a block. Of course, when I declare the array the virtual dictionary pointer may be at some inconvenient

position within a block, so some of our 16-byte items will end up straddling block boundaries. To avoid this, we must align the dictionary pointer, using V.ALIGN. This shifts the dictionary pointer, if necessary, so that it is an exact multiple of the argument passed to it from the block boundary. The offset passed to it must be itself aligned. (This simplifies the code, and there is no good reason to pass it anything other than aligned offsets.)

Once we have a data structure established in the virtual dictionary that observes the aligning conventions, we can access it very simply using V>ADDR, which converts a virtual address into a real address. This it does by turning the virtual address into a block number and offset. It adds the block number to the active base block, loads that block, and adds the offset to the address returned by BLOCK. It is worth remembering that addresses within blocks are of the moment only, and should be used immediately, or the datum copied to some safer buffer. You also need to UPDATE when modifying data.

For what it's worth, I believe blocks should be updated automatically, unless specifically inhibited. That way, instead of being in real danger of losing new data by forgetting to update, you run the less likely risk of losing old data, in the event that having modified a block you decide against keeping the changes and forget to inhibit updating. And, finally, as my mother used to say, don't forget to FLUSH when you have finished.

Addressing Larger Structures

Not every data item will be conveniently less than 1024 bytes long. Equally, you may not be able to afford the wasted space that aligning implies, and it can get pretty horrendous when, say, an item is 513 bytes long and you lose almost half of every block. This can be overcome at the cost of loss of speed (this is one of those classic speed/memory dilemmas that occur almost everywhere in computing).

V@ and V! can be used to access data items that do not observe the block disciplines and are a little more complex. They are, however, very similar. With the exception of a little stack manipulation at the start, they are, in fact, identical but for one word, which in the case of V! needs to be UPDATEd, and for V@ needs to be SWAPped.

Such an unusual circumstance requires unusual factoring, particularly since it is in the middle of a control structure and, as I require good access to four stack items, it is optimal to keep some on the return stack. The route I chose was to effectively patch V! during execution, using the execution vector FIX, which has either UPDATE or SWAP assigned to it, as required.

V! takes three parameters: a source address, a virtual destination address, and a length in bytes of the data to be stored in virtual memory. It then divides the length into handy bite-sized pieces to MOVE into the block buffer. With a large structure, the first and last pieces may not be exactly 1K long, so the first line of code after the WHILE calculates the correct length for each piece. The next line does the MOVE, the third increments the source and destination addresses by the size of the piece that was MOVEd, and the fourth line decrements the remaining length by a similar amount.

Ideally, this would have been done inside a DO...LOOP, except for certain problems with Forth's counted loops. To be specific, V! is best coded with the length as the loop index. This implies a negative-going loop. Furthermore, it should be reasonable to pass a length of zero as an argument, which implies a zero-tripping loop. Unfortunately, zero-tripping loops with a decrementing index do not work correctly. Consider the following definition:

```
: EXAMPLE ( n)
  0 ?DO  ." just once"  -1 +LOOP ;
```

Those who doubt my assertion may wish to work out what argument should be passed to EXAMPLE to cause it to print the string "just once" just once.

V! is used to move data from real to virtual memory. V@ fetches data from virtual to real memory. It takes as arguments a virtual source address, a real destination address, and a length. It returns the address of the buffer to which the data was moved. Sometimes, the most difficult part of coding a word is deciding on its stack effect. This was the case with V@, which accepts arguments like MOVE but returns an address which points to the requested data, sort of like @. This means it does not consume its arguments, as is generally accepted to be good style; but, in practice, this turns out to be a useful arrangement.

Using Virtual Memory

Although we have done what we can to make disk memory look like RAM, it should be borne in mind that this is not RAM, and behaves differently in some respects. In particular, access times can vary. Given, say, a two-dimensional array of integers that is stored on disk in row major format, attempting to index through the array down a column will be a lot slower than traversing along a row, as all the elements in a given row are likely to be on the same block, whereas elements in a given column will be spread over several blocks. Fortunately, as Forth programmers we are in control of our environment, so can anticipate the effects of how our data is organized.

How significant this effect is will vary from Forth to Forth, depending on how many block buffers there are, and what algorithm is used to decide which buffer is written to next. In my case, I have a measly single-block buffer, which is probably the worst of all possible setups; but given one meg. of flat RAM, I seldom use the disk for anything other than saving data at the end of a session.

Those who endure a segmented memory may wish to consider modifying BLOCK so that a range of block numbers refers to chunks of extended memory rather than to disk sectors. This seems a simple way of adapting to an inconvenient architecture.

There is no reason that the virtual dictionary could not contain executable code as well as data. However, this tends to be non-portable, and requires as a minimum the facility to produce headerless words. This means that it is rather beyond the scope of this article, so I simply mention it as a possibility.

Error Handling

The code published here does not contain error-handling routines. This is not idleness, it is a design decision. This decision is made for several reasons, which I will mention in no particular order. Firstly, in published code it distracts from the intention of the article. Secondly, error-handling routines serve two distinct purposes. During program development, they point out the position of bugs in faulty code. It is my experience that faulty code is written because of inadequate understanding of the problem. In this situation, the time spent isolating a bug, once detected, is time well spent, because I come to understand the underlying causes of the problem, which is necessary if I am to rewrite the code correctly rather than just patching a duff routine. Forth's incremental environment makes detecting bugs easy.

During program execution, error handlers are to trap bad data so that program execution can be maintained, or to bomb the program with some fairly useless message such as "Division By Zero Error!" Forth is often used in circumstances where the latter option is not appropriate. The former is best solved by prevention rather than cure. Trapping bad data is best done at the earliest possible opportunity, before it does any damage, rather than letting the inevitable happen and then patching up the pieces. Certainly there are times when it is not possible to trap dangerous combinations of data until the error, and then it is appropriate to use THROW in a low-level routine.

However, what I regard as toolbox routines (I maintain a library of routines I have coded which are applicable to more than one project, such as this code) are developed without error handlers, which I add for particular projects as required. Generally, this is quite easy and means that the error handlers can be tailored to the application in which they will be used.

Finally, I prefer to program without a net, so to speak. I write code just as responsibly without a nagging compiler—such as those that other languages appear to insist on—as with one. Perhaps even more so, as I know it is down to me to get it right, rather than relying on a crutch.

Summary

It is often forgotten, in the Files vs. Blocks debate, that blocks are not some sort of poor man's file system, but represent a window into a virtual memory. They are, admittedly, underdeveloped as a system, but then so are many aspects of Forth. This is not a limitation, but rather allows the programmer to create tools specifically suited to the task at hand, rather than relying on general-purpose tools that cannot be ideal for every situation. Here we have seen one possible way of developing the block system, by mimicking the Forth dictionary-allocation scheme. I doubt that the constraints of a file-based system would allow such an extension to be coded so simply and directly.

Gordon Charlton is the Events and Meetings Secretary of FIG-UK, and contributes regularly to that chapter's newsletter, *Forthwrite*. His last major project was a string-pattern matcher which was presented at euroFORML '91 and '92. If anyone can provide a rigorous description of the Ratcliffe-Obershelp algorithm, he would be pleased to hear from them.

```
\ Virtual  --  ACTIVE  V.PAGE  V.BASE  V.DP  V.HERE  V.ALLOT
1024 constant B/BLK
variable ACTIVE

: V.PAGE ( blk)  CREATE  ,  0 ,
(  )   DOES>  active ! ;

: V.BASE ( --addr) active @ ;

: V.DP ( --addr) active @ 4+ ;

: V.HERE ( --vaddr) v.dp @ ;

: V.ALLOT ( len) v.dp +! ;

\ Virtual  --  V.ALIGN  V.ALIGNED  V>ADDR
: V.ALIGN ( len)   b/blk min  1-  v.here 2dup  and
      IF  or 1+  v.dp !
      ELSE  2drop
      THEN ;

: V.ALIGNED ( len--len)
  0 swap  2* 1 DO  drop i dup  +LOOP ;

: V>ADDR ( vaddr--addr)
  0 b/blk um/mod  v.base @ +  block + ;

\ Virtual  --  V!  V@
variable 'FIX ' update 'fix ! : FIX 'fix @ execute ;

: V! ( addr vaddr len)
  >r BEGIN   r@
  WHILE      b/blk 2dup 1- and -  r@ min
       >r 2dup v>addr fix r@ move
       r@ + swap r@ + swap
       r> r> swap - >r
  REPEAT r> drop 2drop ;

: V@ ( vaddr addr len--addr)
  ['] swap 'fix !  >r dup rot r> v!  ['] update 'fix ! ;
```

("Back Burner," continued from page 43.)
interpreter is sometimes called the *inner* interpreter, in contrast to the *text* interpreter, which is sometimes called the *outer* interpreter.

The code field of a Forth word is a pointer. The parameter field of a word defined with : contains at least one pointer, and may consist of nothing more than a string of pointers.

Although pointers are basic to programming, I never cease to be amazed (likewise, amused) by the treatment of pointers in the C programming language. In that overrated and specious environment, the combination of ill-chosen symbology and convoluted rules makes the application and manipulation of pointers a matter of Byzantine complexity, fraught with peril for tyro and adept alike.

The Ups and Downs of Stacks

The *stack pointer* S holds the address of the top of the parameter stack, i.e., the address at which the most recent stack entry resides. The application programmer typically has no need to access the stack pointer; should, however, the occasion arise, the word 'S has been defined to push the pointer onto the stack.

User variable S0 holds the address of the cell *immediately beneath the base of the stack.* Conveniently, the same address marks the start of the *terminal input buffer,* sometimes called the *input message buffer.* When execution of a word such as QUIT calls for the parameter stack to be cleared, the address in S0 is copied into the stack pointer, whereupon the value *zero* is pushed onto the stack. Thus, in the condition that the stack is *empty,* the stack pointer addresses the cell immediately *above* the cell addressed by S0. An attempt to "dot" the empty stack produces the display

```
. 0 . stack empty
```

Words (such as DEPTH) which access the stack pointer take into account the fact that the empty stack actually has one entry, namely, the *buffer cell* containing the value zero. Presence of a buffer cell containing a known value ensures a predictable and repeatable response to the action of popping or "dotting" an empty stack. The arrangement reflects the fact that the user in the development environment has free access to the parameter stack. Predictable and repeatable responses for erroneous operations reduce confusion and facilitate diagnostics. Note that, although operations performed with erroneous data obtained from stack underflow may cause a system crash, underflow of the parameter stack does not, of itself, corrupt the system.

To see the action of the parameter stack buffer cell, define a word such as

```
: FIZZLE   CR 4 0 DO . SPACE LOOP ;
```

Upon entering

```
1 2 3 FIZZLE
```

...the response will be

```
3 2 1 0 FIZZLE stack empty
```

...the zero being the content of the buffer cell. The entry

```
1 2 FIZZLE
```

...produces (on my system) the response

```
2 1 0 8241 FIZZLE stack empty
```

...the zero being the content of the buffer cell and the value 8241 being the content of the cell *below* the buffer cell.

In embedded applications, the need to conserve RAM generally dictates that the buffer cell be omitted. However, in such an environment, the application program typically guards against stack underflow, making the cell superfluous.

Up the Down Staircase

Now that you see how all this works, allow me to toss in a monkey wrench. Technically, the terms *beneath* and *bottom* are misnomers, considering that, by convention, stacks grow *downward,* i.e., toward low memory. Thus, physically, the terminal input buffer is at a *higher* address than the buffer cell at the base of the parameter stack. The rationale for implementing a stack such that it grows downward is the ease with which the address of a stack item may be developed: one need only *add* a positive integer to the stack pointer.

At this point, you should pencil a diagram in the margin or turn to the diagrams in *Starting Forth* to ensure you understand the arrangement. Incidentally, if you don't already have a copy of *Starting Forth,* by all means obtain one ASAP; a better investment you seldom will make.

Inasmuch as underflow of the return stack is *not* a contingency for which one makes provision (other than a reset button, which should be a part of *every* system), no buffer cell is provided for the return stack. Thus, the first user variable resides immediately below the first cell of the return stack, and the return stack pointer R is initialized to the value residing in the user pointer U. Again, note that, since the return stack grows downward, the physical address of the first user variable is *higher* than the address of the base of the return stack.

The reason no provision is made for underflow of the return stack is akin to the reason airlines provide no parachutes for the passengers: underflow of the return stack is a catastrophic event, the result of a fatal programming error creating a system condition from which recovery is all but impossible.

A copy of the *top* item of the return stack is returned by the word I; a copy of the *second* item on the return stack is returned by the word J.

Your Pad or Mine?

The *pad* ("scratch pad") is of indeterminate size and is located at a fixed offset from the top of the dictionary. Thus, the pad resides in the region between the parameter stack (which grows downward) and the dictionary (which grows upward). The region between the pad and the dictionary is used for formatted numeric output conversion.

The address of the pad is returned by the word PAD, which is defined in terms of the dictionary pointer H.

(Continued.)

Necessary Appurtenances

In order to metacompile, at least two additional pointers are required. Obviously, we need a dictionary pointer for the future application dictionary which will reside in read-only memory (ROM). Additionally, during compilation we need a pointer for the future application read/write memory (RAM), so we can assign RAM for application variables, arrays, etc., and for system functions (stacks, buffers, etc.).

Curiously, it is *not* necessary to have a pointer for the virtual application dictionary which is being built within the development system hardware: the physical addresses within the development system can be computed "on the fly" from the corresponding addresses in the future application dictionary.

We need words to set, to advance, and to read each of the application pointers. We also need a word to transform a future application ROM address into the corresponding address in the virtual application dictionary residing on the development system. Finally, for diagnostic purposes, we require a word to compute the inverse transformation, i.e., a word to deduce the application ROM address from the virtual dictionary address.

In a later column we will choose appropriate names for each of the pointers and for each member within the set of support words.

In Retrospect; a Preview of Coming Attractions; et cetera

A number of these columns may appear to be simply a review of material which has been presented in books such as *Starting Forth*. However, what I am trying to do is illuminate obscure aspects of Forth for which I have found the available documentation to be either incomplete or non-existent. These aspects must be thoroughly understood if one is to attempt his own Forth implementation. Moreover, it is not sufficient merely to know *what* another implementor has done; you must understand *why* he did it in the *manner* in which he did it. Your environment may call for a different technique, or you may be able to devise a superior scheme.

Meanwhile, has anyone attempted to take the assembler presented in column #4 from working prototype to final form? Has anyone built the trainer presented in column #5? Are there any intelligent and attractive single women who would like to date a *Forth Dimensions* author? Does *anyone* out there understand the abbreviation *R.S.V.P.*?

R.S.V.P.

Russell Harris is an independent consultant providing engineering, programming, and technical documentation services to a variety of industrial clients. His main interests lie in writing and teaching, and in working with embedded systems in the fields of instrumentation and machine control. He can be reached by phone at 713-461-1618, by facsimile at 713-461-0081, by mail at 8609 Cedardale Dr., Houston, Texas 77055, or on GEnie (address RUSSELL.H).

("Fast Forthward," continued from page 37.)

face for Computing Standards.)

Once those and the other POSIX services have been shown to be robust and efficient when generated from Forth, Forth can more readily support the many applications that are part of UNIX.

Furthermore, it would be a first step towards open-systems compliance. These POSIX-conformant routines should be incorporated into Forth through a linker's editing of previously compiled, relocatable object modules. Besides following the industry standard for layering software regardless of its source, this would help encourage the POSIX work to be reused across Forth platforms, and would encourage Forth implementations to be more conversant with linkers.

(Forth source code for POSIX routines would also support code reuse, but only if Forth systems have suitable and standard routines for implementation of specific POSIX services. This is not currently possible for services such as process management. Where Forth systems don't address a POSIX feature, the source code is likely to be targeted to a particular Forth rather than the underlying machine on which the Forth is running. By producing precompiled object modules, we make use of a more common, machine-language denominator for code, ultimately leading to more code-reuse potential.)

POSIX services are, ultimately, operating system services. As such they are expected to reside in memory in a machine-language format. The programs that are clients of these services need not be affected by the language of origination. The process of generating POSIX services from Forth will provide us with an opportunity to learn to write code in terms of a more language-independent, layered-software model.

Meanwhile, this installment of the Product Watch *[page 37]* mentions two fascinating Forth systems. One is written in C, and therefore could offer the code reuse potential associated with an open development system. Another Forth system sports a GUI that imitates X/Motif. Someone may even want to use it as the basis of development of a display server, in accord with the X Window System client/server protocol.

Publicity help offered

In the Spring 1993 edition of the MicroProcessor Engineering catalog, we learn of the promotion of Nigel Charig as their Sales and Marketing Manager. MPE focuses on its former customers as the primary readers of this catalog, so the offer on page two may or may not apply to you. In any case, an inquiry wouldn't take much effort. (Refer to the Product Watch section for MPE's address.)

Here's the context in which the offer was made:

"One of Nigel's comments about Forth has been that many people are not using Forth because they don't know about it. Consequently we are planning a series of Forth awareness articles for the electronics and computing press. If you have an application that your company wants free publicity for, contact Nigel so that he can feature it in one of these articles."

On the Back Burner #8

Dead Reckoning

Conducted by Russell L. Harris
Houston, Texas

The collection, cataloging, preservation, and dissemination of knowledge is a matter of concern to all. Even the most cursory investigation is sufficient to reveal that the store of man's knowledge is not nearly as secure as one would expect or desire. On the whole, our knowledge exists, as it were, in fragments residing in scattered modules of battery-backed volatile RAM.

Occasionally we suffer the irretrievable loss of a portion of man's knowledge. At times, financial constraint or negligence allows one of the figurative batteries to expire. On other occasions, duress, sabotage, warfare, or natural disaster results in destruction or deactivation of a memory module. All too often we discover we have no comprehensive backup from which the lost data may be restored.

Lest you think this phenomenon is confined to our present age, consider the testimony of that most authoritative of sages, Solomon:

> That which hath been is that which shall be; and that which hath been done is that which shall be done: and there is no new thing under the sun. Is there a thing whereof it may be said, See this is new? it hath been long ago, in the ages which were before us. There is no remembrance of the former; neither shall there be any remembrance of the latter that are to come, among those that shall come after.
>
> —Ecclesiastes 1:9–11

Out of Sight, Out of Mind

The event which set my mind onto this train of thought was the reading of an article announcing the closing of what was described as the foremost engineering library in the world. Located in New York city, the library suffers from inadequate funding. The vast collections of the library are being dissipated throughout the land, to other libraries and organizations. It is almost certain that such a comprehensive collection will not again be assembled in one location.

After knowledge is collected and catalogued, there remains the matter of preservation. I recall several years ago reading of a technique developed by NASA to arrest the deterioration of books printed on acid-base paper. It seems that even the portion of man's store of knowledge which, for the purpose of preservation, has been typeset, printed, and bound, is slowly being obliterated as atmospheric moisture activates the acid residue from the paper-making process.

An Obiter Dictum Regarding the Obfuscation of Abbreviation

In every phase of its existence, from initial recording to ultimate retrieval, knowledge is subject to corruption. One mechanism by which knowledge is obscured and obliterated is the insidious and damnable practice of abbreviation. As an example, consider the term *dead reckoning*. Even Webster, apparently, is unaware that the term is actually an abbreviation: the lexicon only indirectly sheds light on the etymology, defining *dead reckoning* as "determination without the aid of celestial observations of the position of a ship or aircraft *deduced* from the record of courses sailed or flown, the distance made, and the known or estimated drift." The fact is that the word *dead* in the term *dead reckoning* is simply a mispronunciation of the abbreviation *ded*. Thus, indolence and carelessness, expressed through abbreviation and mispronunciation, respectively, have given birth to the nonsensical term *dead reckoning*.

Having survived this brief lamentation and etymological digression, the astute reader will recall that we are in the process of examining metacompilation, an activity which, as you by now may have deduced, calls for a considerable amount of dead reckoning. Whereas navigators reckon in terms of geographical coordinates, programmers reckon in terms of addresses within the address space of a computer. Navigators deal with speeds and headings; programmers deal with quantities of data and directions of increment.

Addressing the Matter of Pointers

Forth is a language of *pointers*. In the following paragraphs, by *pointer* I mean simply a register or a memory location which holds an address. However, it is not uncommon to see the word *pointer* used for the address itself. Context usually makes clear the usage.

The pointer of which most programmers are first aware is the *dictionary pointer* `H`, which holds the address of the next available byte in the dictionary of the development system. The value in `H` is returned by the word `HERE`.

The *virtual Forth machine* is based upon a set of pointers, which are as follows:

`U`	the *user* pointer; holds the base address for *user* variables for the current user (i.e., task)
`S`	the parameter *stack* pointer
`R`	the *return* stack pointer
`I`	the address *interpreter* pointer; holds the address of the code field of the next word to be executed
`W`	the address interpreter *word* pointer; holds, upon entry to executable code, the address of the parameter field of the word being executed

Although variables are frequently used for pointers, this particular group of pointers is typically implemented in processor registers, in order to minimize access time and thus maximize speed of the Forth machine. Note that the *address*

(Continues on page 41.)

CALL FOR PAPERS

for the fifteenth annual and the 1993

FORML CONFERENCE

The original technical conference
for professional Forth programmers, managers, vendors, and users

Following Thanksgiving
November 26–November 28, 1993
Asilomar Conference Center
Monterey Peninsula overlooking the Pacific Ocean
Pacific Grove, California U.S.A.

Theme: **Forth Development Environment**

Papers are invited that address relevant issues in the establishment and use of a Forth development environment. Some of the areas and issues that will be looked at consist of networked platform independence, machine independence, kernel independence, Development System/Application System Independence, Human-Machine Interface, Source Management and version control, help facilities, editor development interface, source and object libraries, source block and ASCII text independence, source browsers including editors, tree displays and source data-base, run-time browsers including debuggers and decompilers, networked development-target systems.

Additionally, papers describing successful Forth project case histories are of particular interest. Papers about other Forth topics are also welcome.

Mail abstracts of approximately 100 words by September 1, 1993.
Completed papers are due November 1, 1993.

We anticipate a full conference this year.
✔ Priority will be given to participants who submit papers.
✔✔ Earlier papers will be given preference in choosing longer presentation periods

John Hall, Conference Chairman

Robert Reiling, Conference Director

Information may be obtained by phone or fax from the
Forth Interest Group, P.O. Box 2154, Oakland, CA 94621. 510-893-6784, fax 510-535-1295
This conference is sponsored by FORML, an activity of the Forth Interest Group, Inc. (FIG).

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, all meals, and nightly wine and cheese parties.