$10 Volume XVI, Number 6 March 1995 April

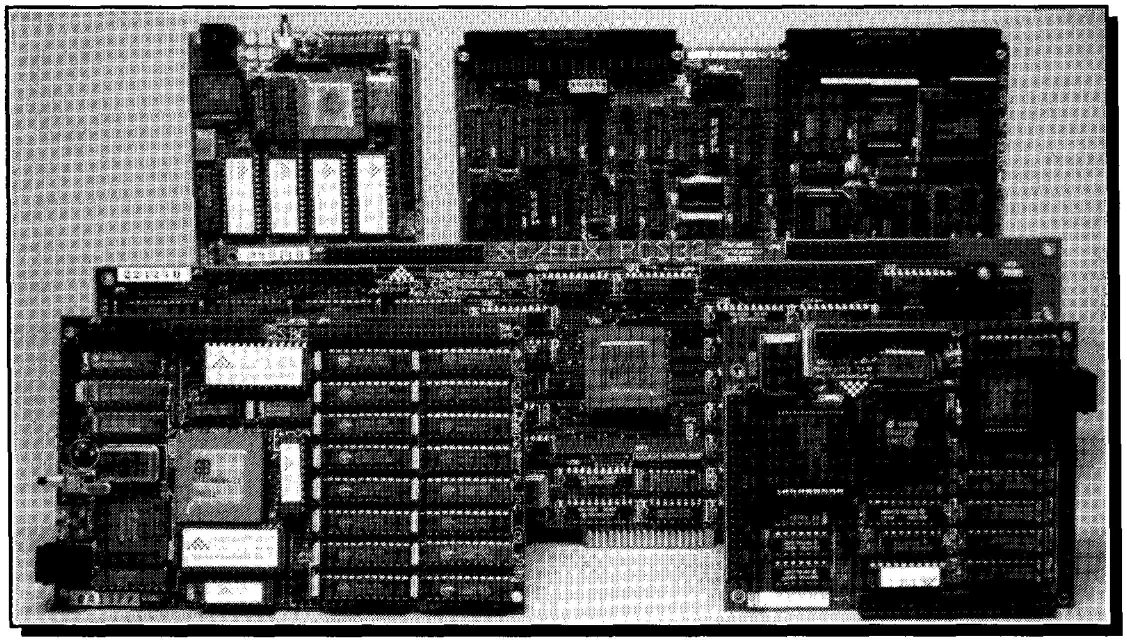
FORTH DIMENSIONS

SILICON COMPOSERS INC

FAST Forth Native-Language Embedded Computers

DUP

>R



C@

R>

Harris RTX 2000tm 16-bit Forth Chip
- •8 or 10 MHz operation and **15 MIPS** speed.
- •1-cycle 16 x 16 = 32-bit multiply.
- •1-cycle 14-prioritized interrupts.
- •two 256-word stack memories.
- •8-channel I/O bus & 3 timer/counters.

SC/FOX PCS (Parallel Coprocessor System)
- •RTX 2000 industrial PGA CPU; 8 & 10 MHz.
- •System speed options: 8 or 10 MHz.
- •32 KB to 1 MB 0-wait-state static RAM.
- •Full-length PC/XT/AT plug-in (6-layer) board.

SC/FOX VME SBC (Single Board Computer)
- •RTX 2000 industrial PGA CPU; 8, 10, 12 MHz.
- •Bus Master, System Controller, or Bus Slave.
- •Up to 640 KB 0-wait-state static RAM.
- •233mm x 160mm 6U size (6-layer) board.

SC/FOX CUB (Single Board Computer)
- •RTX 2000 PLCC or 2001A PLCC chip.
- •System speed options: 8, 10, or 12 MHz.
- •32 KB to 256 KB 0-wait-state SRAM.
- •100mm x 100mm size (4-layer) board.

SC32tm 32-bit Forth Microprocessor
- •8 or 10 MHz operation and **15 MIPS** speed.
- •1-clock cycle instruction execution.
- •Contiguous 16 GB data and 2 GB code space.
- •Stack depths limited only by available memory.
- •Bus request/bus grant lines with on-chip tristate.

SC/FOX SBC32 (Single Board Computer32)
- •32-bit SC32 industrial grade Forth PGA CPU.
- •System speed options: 8 or 10 MHz.
- •32 KB to 512 KB 0-wait-state static RAM.
- •100mm x 160mm Eurocard size (4-layer) board.

SC/FOX PCS32 (Parallel Coprocessor Sys)
- •32-bit SC32 industrial grade Forth PGA CPU.
- •System speed options: 8 or 10 MHz.
- •64 KB to 1 MB 0-wait-state static RAM.
- •Full-length PC/XT/AT plug-in (6-layer) board.

SC/FOX SBC (Single Board Computer)
- •RTX 2000 industrial grade PGA CPU.
- •System speed options: 8, 10, or 12 MHz.
- •32 KB to 512 KB 0-wait-state static RAM.
- •100mm x 160mm Eurocard size (4-layer) board.

For additional product information and OEM pricing, please contact us at:
SILICON COMPOSERS INC 655 W. Evelyn Ave. #7, Mountain View, CA 94041 (415) 961-8778

Contents

Editorial

The FIG office recently received a letter from someone who does not intend to renew his membership/subscription. He states, "After reading the last several months [of] Forth Internet usenet banter, it seems that the principal players have positioned themselves as erudite adversaries. This does not bode well for Forth's popular growth and probably indicates its demise."

(Here I should point out that comp.lang.forth is an unmoderated USENET newsgroup. Like other unmoderated newsgroups, it has a public life and character of its own that is dictated by those who post to it, but which is perceptible by all its on-line "lurkers.")

I agree that some of the more politicized of the on-line threads have been distastefully polemical and personal, to say nothing of unprofessional. I do not believe these represent Forth's principal players, though. I don't blame anyone for wanting to avoid such scenes, but they don't serve as indicators of Forth's demise.

The real leaders in the Forth community are running Forth businesses, creating Forth systems and products, and providing services. They are honing their expertise, positioning Forth in the marketplace, and adopting contemporary programming practices in creative, Forth-like ways.

I hope the new leaders of the Forth Interest Group—the incoming Board of Directors and FIG chapter leaders—will come from the ranks of Forth's real "key players."

If you are interested in affecting the future of the Forth Interest Group, there is no time like the present. If you or another likely candidate is not among those listed in the forthcoming announcement of Board nominees, you can have your name placed on the list of candidates—for details of the procedure provided in FIG's by-laws, see the full text of the announcement in our last issue; if you have questions after reading it, contact current FIG Board members for clarification and to declare your interest in a Board position.

* * *

People regularly ask how to learn Forth. In this issue, Richard Fothergill's letter prompted me to remember how some Forth experts opine that techniques can be shared, but realizations can only be hinted at. This contributes to the frustration of outside observers, causing some to say Forth is too mystical to be practical. (E.g., a Forth programmer's productivity cannot be measured by the number of lines of code produced per day.)

This also challenges the writers of Forth tutorials, the instructors of Forth courses, and the marketing expertise of Forth vendors. It is not enough to teach the use of stack operators and wordlists and defining words, the student must be prepared and led methodically through successive stages of understanding.

One hopes the coming, new generation of Forth tutorials and texts will achieve this. To do less is to ask readers to jump through seemingly insignificant hoops ("hello, world") for no obvious purpose, or to suffer through slick prestidigitation that dazzles without illuminating. We must generate Forth expertise—by defining the entry points from which newcomers approach Forth and structuring our educational efforts accordingly, and by better understanding and teaching the nuances that comprise "Forth thinking."

* * *

Speaking of Forth wizardry, I am pleased to welcome Wil Baden as *FD*'s newest columnist. Wil is well known at FORML conferences as both astute and entertaining, a very enjoyable speaker and accomplished Forthwright. (And when preparing to join the FORML lecture tour of China some years ago, he learned to speak Chinese well enough to deliver his talks without a translator.) Wil's list of proposed topics for "Stretching Forth" is impressive—we look forward to his contributions, which begin in this issue.

—Marlin Ouverson
FDeditor@aol.com

Forth Dimensions
Volume XVI, Number 6
March 1995 April

Published by the
Forth Interest Group

Editor
Marlin Ouverson

Circulation/Order Desk
Frank Hall

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

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

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

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

Letters

Forth's Northern Exposure

Hello,

I was a member of FIG earlier, but $52 for a poor student was too much.

Now I work at an electronics company which I try to convince to use Forth. It's tough, with all the C programmers, but I have convinced our C master. So now we are constructing a development system that will be interactively programmed in Forth.

I see Forth as a perfect language for embedded systems, and believe that most other people will think so, too, after using this sytem. But maybe it would help if there were a smoother transition from C to Forth. I have used a Forth system from Triangle Digital Services, who I heard of through an advertisement in *Forth Dimensions.* In the latest news from TDS, they write about C-to-Forth articles in *FD*, so now I am joining FIG again.

Best regards,
Anders Eriksson
Mölndal, Sweden

Anders, we are pleased to welcome you back as one of our readers, and we also thank you for supporting both Forth and its commercial vendors. Let us know how your project progresses!

—Ed.

On the Learning Curve

Dear Marlin Ouverson,

I am a newcomer to the Forth programming language who has recently acquired a copy of Tom Zimmer's F-PC v. 3.55. I have discovered that your journal, *Forth Dimensions*, deals with matters relating to Forth, and I am writing to ask for more specific information about your publication.

Firstly, I would like to know if *Forth Dimensions* would be suitable for a beginning Forth programmer who is a hobbyist, rather than a professional. Although I have read Leo Brodie's book, *Starting Forth*, and understand some of the basics of Forth such as the stack operations, creating new words, etc., I haven't yet got to grips with the more arcane topics, such as creating separate vocabularies, vectored execution, etc.

Secondly, the latest information I have is that *Forth Dimensions* costs $52 to overseas subscribers. Is this correct?

Thirdly, since it is a long row to the States from Wales, do you have an advice column that is contactable via the Internet?

And lastly, do you have a back issue or two available that I could look at so that I could see the level and style of articles?

I'd be very grateful for any help you can give me on the above.

Yours sincerely,
Richard Fothergill
Mid Glamorgan, Wales
United Kingdom

Thank you—your questions are important to anyone *new to Forth, so I'll make space here for some answers.*

As other writers have noted, Forth's unique paradigm can make the learning process a bit of a challenge. Many of today's experts describe their introduction to Forth as a process of gathering bits and pieces of information and techniques until an elusive understanding suddenly dawned on them, more like a realization than a linear education. This may indicate a need for better tutorials—and Forth Dimensions *always welcomes such material—but it also points out that Forth encourages a different way of thinking about computer-based problems and solutions. And, as András Zsótér points out in this issue, the understanding gained via Forth often benefits one's work in other languages.*

To answer your first question more directly, newcomers to Forth are sometimes intimidated by the content of Forth Dimensions. *Those who persevere do find it rewarding, and we will do our best to include material helpful to you and other beginners. Try to augment your reading of* Starting Forth *with the same author's* Thinking Forth *and, perhaps, material like Haskell's* The Forth Course *and Ting's* The First Course *(specifically geared toward users of F-PC). Such tutorials will prepare the foundations, while material from* FD *and other sources will stretch your understanding and serve to demonstrate what lies just over the horizon of your current knowledge.*

(And yes, overseas memberships/subscriptions are $52. In addition to receiving this journal, FIG members enjoy a discount on books and other items.)

Many Forth experts are available via the Internet. I recommend suscribing to the comp.lang.forth newsgroup, where seasoned Forth users enjoy assisting newcomers, discussing technical matters, and occasionally engaging in political diatribes. Forth is also an emerging presence on the World Wide Web.

For copies of past Forth Dimensions, *membership, and mail order items, see the mail-order form in this issue. We welcome your interest and your questions; please let us know how we can be of further assistance.* *—Ed.*

Tool Interactivity for Rapid GUI Application Development

Gary Ellis, Roy Goddard
Southampton, England

In a high-pressure, commercial environment, rapid development of GUI applications is vital. This paper demonstrates that careful design decisions and algorithm choices are essential to support the tools required. If these choices are made well, the tools produced can interact transparently with each other and the user to allow cost-effective commercial application development.

Introduction

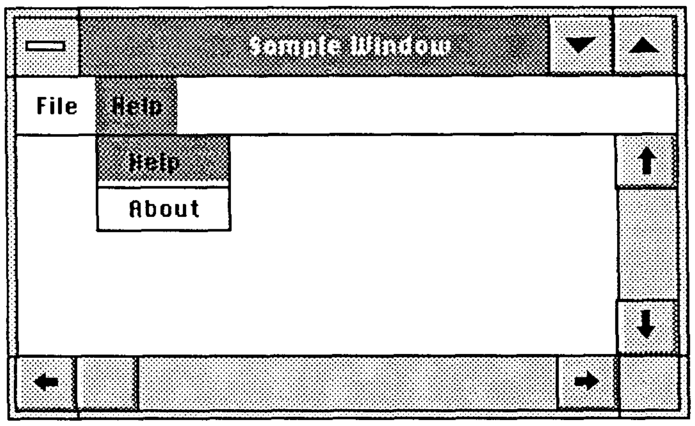
This paper focuses on the MPE ProForth GUIDE tool. It shows how careful design decisions and algorithm choice can allow a Forth utility to provide GUI-friendliness and yet lose nothing of the interactivity we take for granted in Forth. It shows how the interactivity extends from the user/system interaction to the tool/user/application interactions which are vital for fast application development in a high-pressure commercial environment.

Feature Overview

Before describing the design and algorithm choices associated with a tool as complex as ProForth GUIDE, it is useful to look at the features of the tool, to see where the design decisions and algorithm choices fit in—why they are important.

The specification for GUIDE requires that the user is able to describe and define a window and or a menu without typing any lines of code. It also requires that the user be able to define a dialog without writing any lines of code. The code generated by GUIDE must be human-readable, and conform to the ProForth language description in the system manuals. It must also be able to be read back into GUIDE for the user to continue graphical work on it. Between sessions using GUIDE, the user must also be able to use a standard text editor to work on the output source files with minimal coding restrictions to re-use GUIDE on the same files. GUIDE also creates the complete application framework, including window procedures and stub code for all specified events and messages able to be processed or generated by the control, dialog, menu, or window.

This specification defines a product which compares favorably with other GUI builders, such as Microsoft Visual C++, Blue Sky Visual Programmer, etc. However, the specification is tighter, in that the code generated must be very human readable without learning new or private libraries and classes, and requires no additional DLLs, etc., for the code to compile and run in an application.



User Interface—Design Principles

The major design principle for the user interface is the use of the mouse for every possible action. The example included in this paper describes a window with a four-item menu. The only text typed in for the entire description is the text to appear in the menu. All other specifications and requirements of the application window were performed with just the mouse. Further, as only a live demonstration can adequately show, most actions are centered around the sample window. This is changed by using the mouse and the left button where Microsoft has a convention (such as dragging), and the right button for other things. The use of the right button to open a dialog of object properties is fast becoming a standard under Windows.

Although the code for GUIDE itself may take longer to develop, the usability advantages are enormous. It is this usability which gives the Forth/GUIDE user the development edge, and more opportunity to exploit his or her own application market.

A further design decision for the user interface is the ability to define dimensions and locations, etc., in two ways. Taking the example of moving a window about on the screen, the first reaction of the user is simply to drag the window to the required location. However, the mouse movement may not be accurate enough, especially on a notebook computer. Therefore, the user is also able to type exact coordinates into the relevant dialog. Thus, there is a symmetry in the development process—either view GUIDE as a graphical editor, or as a property browser through standard source code.

Data Structures—Design for Flexibility

GUIDE is heavily data oriented. Every object which the user can define has at least five of its own, specific properties—such as x- and y-coordinate, width, height, text caption, etc. If the object is a window, it also has attributes of style, extended style, name, messages to process, etc. Thus, if GUIDE is to allow the user to define more than one dialog or one window/menu at a time, the data structures have to be very versatile.

The data structures within GUIDE can be viewed as a tree of linked lists. The best example of this is the menu. A menu contains one object per entry, and there is a list of entries. However, if an entry has a child menu associated with it, this child menu is the head of a new linked list of entries in the child menu. Dialogs, similarly, rely on this kind of structure, as the outer window has child controls, and each control has a list of messages and events to respond to. As many objects have similar properties, coordinates for example, these are in the same place in each structure, so that a certain degree of object orientation is achieved. This makes the structures in GUIDE rigorous, and the tool itself relatively easy to develop.

Code Write-out—Human-readable Code

Most equivalent tools on the market today require libraries and classes such as the Microsoft Foundation Class. Therefore, the code generated refers to these code sources and the user must learn yet another tool in order to understand or modify the code generated by the GUI tool. A vital design decision for GUIDE was to ensure that the machine-generated code used only calls to words and data structures defined and documented in the ProForth manuals and glossary. Therefore, the tool is simply machine-generating code, and not "cheating" by hiding functionality in a private library.

This is very important, as it is possible for the user to understand the code generated and to use GUIDE not only as a productivity tool but also as a learning aid to writing well-structured Forth code. It is also more efficient for the user to debug the resulting code, as he has full access to the functions in use. He can use his standard debugging techniques with the machine-generated code. The alternate tools rely on a debug version of the DLL, which might, of course, differ from the run-time version.

Code Read-back—Interactivity is Vital

Forth is an interactive language. Applications are developed bottom-up and interactively. A tool such as GUIDE generates a complete, proven, application framework, which therefore needs less bottom-up testing. However, the useful application code must be tested and attached to the relevant messages and events. The user will, therefore, be using an editor such as WinTed or another standard text editor to make these modifications and additions. It is vitally important that the code written by the user does not interfere with the machine-generated code, and vice versa. It is also important that the user knows what he may change and what he may not. The tools used must not interfere with each other, and a change made by GUIDE to a source file must not 'break' code written by the user. Therefore, accurate and thorough code read-back is an important aspect of the design of GUIDE. This section discusses the decisions made in this respect.

Traditional Forth Parser

The traditional parser technique in Forth is to use a vocabulary-based parser, with each word in the target language to be matched by a word in a "parser" vocabulary which might set flags, set data, etc.

This kind of parser requires a very specific hard-coded language definition, and may not be appropriate if there is much forward-scanning of syntax. It is also difficult to implement further language structures, as the parser structure is very tightly coupled with the data output or code generation. Terminators also present a problem, as the standard Forth parse requires a white-space character to terminate a word, line, or page. However, the language may need to parse text out of the name of a word and not just use the entire name of the word. This traditional parser is used in MPE's XREF3 product, where there is no implicit structure in the objects being parsed. This is not the case, however, with GUIDE.

BNF Parser

The traditional parser was looked at and turned down for GUIDE. Instead, a BNF parser was implemented. This was based on an equivalent system developed by Brad Rodriguez [1]. Beside exceeding the limitations of the vocabulary-based parser, the BNF parser offers a formal language grammar. This reduces the number of special cases in the parser, and therefore increases the reliability of the system. It is also, therefore, possible to easily extend the parser or change it as the underlying target language changes. It is also possible to define grammars and syntax for more than one input language with no reliance on word order, etc.

The use of a versatile parser such as the BNF one allows GUIDE to accurately parse the user's source code. This, therefore, allows the user to make modifications to the machine-generated code and not render GUIDE useless from that point on.

Page-oriented Chunking

There are two kinds of user-editable code in the GUIDE machine-generated source file. The first of these is code definitions where the user simply edits the body of a definition but GUIDE defines the name of the word. Such words are typically the actions upon receipt of a specific message by the winproc.

The second type is code totally and freely defined by the user, perhaps only called by the body of a word which is the action of a message or event.

There is also a third kind of code: code which may only be edited by the user if obeying very strict rules.

Each of these kinds of code are produced by GUIDE on different pages of the output file. For a discussion of pages in a source files, see Pelc/Waters [2]. Each page is identified by an identification letter. This is parsed by the BNF parser to tell whether to expect user-defined code in the coming page. Thus, the system does not have to buffer text more than necessary. Again, standard tools allow the GUIDE system to interact with both the user and other tools.

Summary

Tools such as GUIDE are designed to allow the user to develop an application interactively. This extends to interaction between the tools themselves. Therefore, the tools can be used for maximum productivity. However, the design decisions and algorithm choices must be made very carefully in order to allow maximum usability and interaction.

References

[1] "A BNF Parser in Forth," Bradford Rodriguez. *ACM SIGForth Newsletter*, Vol. 2, No. 2, December, 1990.

[2] "A Text File Syntax for Screen File Users," S. Pelc and A. Waters. *EuroFORML '88*, Southampton U.K.

This paper was originally presented at the EuroForth '94 conference, Winchester, England.

Gary Ellis (B.Sc., Hons., in Computer Science from the University of Kent at Canterbury) is a Technical Support and Quality Assurance Engineer at MicroProcessor Engineering Ltd., where he has worked for about four years. He regularly teaches Forth to beginners, and was the software engineer responsible for about 90% of the code for the GUIDE tool included in ProForth for Windows. His current goal is to prove that software-engineering principles can be applied to Forth as well as they can to anything else. Mr. Ellis can be reached at gie@mpeltd.demon.co.uk via e-mail.

Roy Goddard (B.Sc. in Mechanical Engineering from the University of Southampton) is a Senior Engineer at MicroProcessor Engineering Ltd., for whom he has worked for seven years, and was the lead programmer on ProForth for Windows. He has used Forth for a wide variety of embedded systems and PC applications over the years and, naturally, feels that Forth is the best thing since sliced bread. Mr. Goddard is available at the rgg@mpeltd.demon.co.uk e-mail address.

ProForth for Windows, which includes GUIDE, is sold and supported in the U.S. and Canada by Forth, Inc. (announced in the preceding issue of *Forth Dimensions*).

Example machine-generated code.

Following is code generated by GUIDE for the example window shown above.
NB. 1. This code modified for brevity. Please ask for full copy.
2. The sequence (p) is used here to denote a page-break (12H) character

```
(p)\ GUIDE Z -- NONE -- Mon 24/10/1994 12:17:54 pm

[forth]

"button" constant "pushbutton"      "button" constant "radiobutton"
"button" constant "checkbox"        "button" constant "groupbox"

"scrollbar" constant "hscrollbar"   "scrollbar" constant "vscrollbar"

(p)\ GUIDE A -- Window#0 -- Mon 24/10/1994 12:17:55 pm

Window (Window#0)              \ data struct for window

(p)\ GUIDE C -- Window#0 -- Mon 24/10/1994 12:17:55 pm

2 Constant ID#2         5 Constant ID#5
3 Constant ID#3         7 Constant ID#7

(p)\ GUIDE D -- Window#0 -- Mon 24/10/1994 12:17:55 pm

2 Ladder Menu: Menu#1
  Subs:
  End-subs
  MF_STRING ID#2 z" New" Text:
```

```
  MF_STRING ID#5 z" Exit" Text:

3 Ladder Menu: Menu#2
  Subs:
  End-subs
  MF_STRING ID#3 z" Help" Text:
  Separator
  MF_STRING ID#7 z" About" Text:

2 Bar Menu: Menu#0
  Subs:
   Menu#1
   Menu#2
  End-subs
  z" File" Submenu: Menu#1
  z" Help" Submenu: Menu#2


(p)\ Application code -- Window#0

(p)\ GUIDE K -- Window#0 -- Mon 24/10/1994 12:17:55 pm

: Menu#1-ID#2           \ hwnd msg wparam lparam -- status ; stub for New
  windefwindowproc      \ replace this with your action
;

: Menu#1-ID#5           \ hwnd msg wparam lparam -- status ; stub for Exit
  windefwindowproc      \ replace this with your action
;

: Menu#2-ID#3           \ hwnd msg wparam lparam -- status ; stub for Help
  windefwindowproc      \ replace this with your action
;

: Menu#2-ID#7           \ hwnd msg wparam lparam -- status ; stub for About
  windefwindowproc      \ replace this with your action
;

(p)\ GUIDE L -- Window#0 -- Mon 24/10/1994 12:17:55 pm

: Window#0-WM_COMMAND   \ hwnd msg wp lp -- status ; menu handler
  over $FFFF and        \ ID in low word of wparam
  case
    ID#2    of Menu#1-ID#2    endof
    ID#5    of Menu#1-ID#5    endof
    ID#3    of Menu#2-ID#3    endof
    ID#7    of Menu#2-ID#7    endof
    drop WinDefWindowProc
  end-case
;

 (p)\ GUIDE M -- Window#0 -- Mon 24/10/1994 12:17:55 pm

: Window#0-FP_CREATE    \ hwnd msg wp lp -- status ; stub to call
  windefwindowproc      \ replace this with your action
;
```

(Continues...)

```
: Window#0-WM_VSCROLL   \ hwnd msg wp lp -- status ; stub to call
  windefwindowproc      \ replace this with your action
;

: Window#0-WM_HSCROLL   \ hwnd msg wp lp -- status ; stub to call
  windefwindowproc      \ replace this with your action
;

: Window#0-WM_CLOSE     \ hwnd msg wp lp -- status ; stub to call
  windefwindowproc      \ replace this with your action
;


(p)\ GUIDE N -- Window#0 -- Mon 24/10/1994 12:17:55 pm

: Window#0-winproc      \ h msg wp lp -- status ; message dispatcher
  2 Pick                \ message
  Case
    FP_CREATE     of Window#0-FP_CREATE   endof
    WM_COMMAND    of Window#0-WM_COMMAND  endof
    WM_VSCROLL    of Window#0-WM_VSCROLL  endof
    WM_HSCROLL    of Window#0-WM_HSCROLL  endof
    WM_CLOSE      of Window#0-WM_CLOSE    endof
    Drop WinDefWindowProc
  End-case
;

Assign Window#0-winproc To-WinProc (Window#0)

(p)\ GUIDE O -- Window#0 -- Mon 24/10/1994 12:17:56 pm

: Window#0         \ -- ; word to execute window
  WS_CAPTION WS_HSCROLL or WS_MAXIMIZEBOX or
  WS_MINIMIZEBOX or WS_POPUP or WS_SYSMENU or WS_THICKFRAME or
  WS_VSCROLL or
  Menu#0 handle                   \ menu name
  "" Sample Window" $>asciiz      \ caption
  #233 #164 #342 #197             \ x, y, width, height
  (Window#0) styled-menu-popup    \ create and show
;
```

(MuP21, continued from page 43.)

to improve on the RISC architecture. By insisting on the minimum set of instructions, microprocessors can be further simplified and their performance improved. We were amazed that MuP21 can run at a peak speed of 100 MIPS using the currently outdated 1.2 micron CMOS process. With the more advanced 0.8 micron process, MuP can be made to run at 200 MIPS rate. Moving on to 0.5 micron, the speed can be increased further to 300 MIPS without much effort.

MuP21 is a 20-bit microprocessor, constrained by the 40-pin DIP package. Using packages with more pins, the design can be easily expanded to 32-bits and beyond. A wider data/address bus will improve the throughput and also allow greater addressable memory space for applications dealing with massive amounts of data. This is another direction in which to evolve the MISC architecture.

With a simpler and more efficient architecture, MISC processors can be built with smaller silicon dies and, thus, the yield will be much higher than for the more complicated RISC and CISC processors. The MISC processors will also consume much less power when running at equivalent speeds. MISC processors will be much cheaper than RISC and CISC processors, and can compete effectively against them on the basis of a favorable price/performance ratio.

An Assembly Programmer's Approach to Object-Oriented Forth

András Zsótér
Hong Kong

Introduction

Some people use Forth as a computer language, while other people—including myself—use it as a computer program for controlling their machine and other pieces of hardware. When I had to decide what to use in our project (a kind of laboratory automation with robotics), I did not search for a computer language (I feel enough at ease with assembly and Pascal to do the job), but I searched for a system which would give me the freedom to do whatever I wanted. I also preferred an interactive program to a compiler generating standalone applications. Naturally, the solution was a Forth system.

On the other hand, during my previous pieces of work I used the object-oriented facilities of Turbo Pascal and felt I would miss that if I had to use a language without it. As an obsessed assembly programmer, I decided to implement a version of Forth for myself and I ended up with a system which is very convenient to use if one wants to be sure all the time what is going on in it.

Basically, I shaped my Forth system after the old-fashioned fig-Forth[1] with some modifications. Because the program runs on a 486 machine, the most natural solution was to use its 32-bit protected mode. This way, I do not have to worry about running out of address space. Also, protected mode really means some protection against accidental mistakes and their consequences.[2]

"A programmer who is not willing to use proper technique cannot be forced to do so."

Definition of Compile-Time Behaviours

Because I am very much concerned about the speed of my program, I decided to generate native code. The technique I used for code generation is sometimes mentioned in the Forth literature as nano-compiling [1]. While older Forth compilers used the CFA to store the address of a machine code subroutine to be called when a word is being executed, I used an additional one (CCFA, the compile-time code field address) to store the address of the subroutine to be called when a word is being compiled. The user of the program can explicitly define the compile-time action of a non-immediate word by using the words `C:` and `;C`.[3] For example, one might want to implement `SWAP!` in the following way:

```
: SWAP! ( Addr Data -- )
    SWAP ! ;
    C: POSTPONE SWAP POSTPONE ! ;C
```

Alternatively, more optimized machine code can be generated if someone has a more-intimate knowledge about the system. Immediate words have the same routine for compile-time and run-time behaviour. This way, the user has control over the code generation and can specify explicitly which routines are to be in-lined or substituted by more adequate machine instructions, and which are to be left alone and treated as ordinary subroutines. One might complain that the old-fashioned, state-smart words can do the same, and the previous example might have been coded as:

```
: SWAP! ( Addr Data -- )
    STATE @ IF
    POSTPONE SWAP POSTPONE !
    ELSE SWAP ! THEN ; IMMEDIATE
```

This is true as long as words such as `COMPILE`, `[COMPILE]`, `COMPILE,`, and `POSTPONE` do not mess up everything.[4] With the use of CCFA,[5] the definitions of compiling words became almost trivial.

`POSTPONE` *Name* generates a call to the CCFA routine of *Name*. `COMPILE,` interprets the top item on the stack as a CFA ("execution token") and, if the word is immediate

1. At the present stage, the program has an ANS-compatible mode which supports most of the features of the new standard.
2. I needed full control, but I did not need too much operating system connection. So I implemented a V86 monitor which takes care of file I/O and other operating system connections by running DOS in a virtual 8086 machine and provides facilities to execute a 486-style (32-bit), protected-mode program.
3. The default action is to generate a subroutine call to the body of the word.
4. For example, consider the effects of `POSTPONE SWAP !` in the latter case. Is this the semantics one might expect?
5. In ANS Forth terms, CCFA would be called a "compilation token."

Figure One. Words that manipulate the pointer to the active object.

```
O!     ( Object --     ) Select object = Write the object pointer.
O@     ( -- Object     ) Query object  = Read the object pointer.
O>R    (R: -- Object ) Save object pointer to the return stack.
R>O    (R: Object -- ) Retrieve saved object pointer from the stack.
{      ( Object1 --    ) (R: -- Object2) This word combines the
                                     functionality of O>R and O!.
}      (R: Object -- ) The same as R>O.
```

(its CFA and CCFA point to the same address), COMPILE, generates a call to that address. If the word is not immediate, COMPILE, simply calls the routine pointed to by CCFA (i.e., compiles the run-time behaviour of the word into the new definition). [COMPILE] and COMPILE can be defined the following way:

```
: [COMPILE] ( -- )
    '  COMPILE,
   ; IMMEDIATE

: COMPILE   ( -- )
   ' LITERAL POSTPONE COMPILE,
   ; IMMEDIATE
```

One Word with Multiple Names

Forth programmers tend to "factor out" similar pieces of code in their programs. As the Forth language grew bigger and bigger, pieces of code appeared with the same effect under different names. If one uses words implemented by different programmers, it is good sometimes to have all the usual names ready. The most common Forth solution is the definition of the new name in the form:

```
: NewWord OldWord ;
```

The above solution is usually satisfactory; however, if one wants *NewWord* to behave *exactly* as *OldWord*, a more sophisticated definition is needed:

```
: NewWord OldWord ;
C: POSTPONE OldWord ;C
```

If *OldWord* is immediate, the definition is different:

```
: NewWord
   POSTPONE OldWord ; IMMEDIATE
```

To avoid all this trouble, a new definition word `Alias ( CFA -- )` is provided. So from now on, the above definition would be:

```
' OldWord Alias NewWord
```

This definition will work, regardless of the immediacy of *OldWord*. At first sight, this facility does not seem to be of much help but, because of the OOP facilities, it is sometimes necessary to have a name ready in multiple vocabularies. Also, Forth programmers name their words on a pragmatic basis (what the word is used for) and not on a semantic one (what is the effect of the word). If a piece of code has multiple usages, the use of aliases can greatly increase the readability of the program.

What is an Object?

One of the most powerful features of my Forth implementation is that it is object oriented. There are several problems with this utterance. Some people even argue that Forth, in itself, is an object-oriented language because of the CREATE ... DOES> capabilities. In my opinion, a real OOP is more sophisticated than that, and polymorphism, inheritance, and virtual methods[6] are necessary in a system to qualify it as an OOP language. In my interpretation, an object is an entity which consists of data (residing at least partially in memory) and a set of methods to manipulate the data. I always considered an object to be quite independent from its environment. In order to make the latter explicit, I introduced the idea of the active object. Only one object can be active at a time. The system has a pointer to the active object,[7] which can be manipulated via the words in Figure One.

As the term "active" already implies that an object is considered to be an "animate" entity, this means individual behaviour is attributed to each and every object instance, a reason why OOP is called "programming in the active voice" [2].

An object can access the application's memory in two ways. The first way is the same as we normally address the memory, and the second one is when all addresses are relative to the object's base address. At first sight, this latter way seems to make sense only for the fields of an object (so that they are represented as "offsets" from the starting address of the object), but it can be useful also for addressing other entities outside the object.[8] In order to make the data stored in an object accessible for the traditional Forth memory operations such as @ and !, a new word is introduced. This new word is `↑ ( RelAddr -- AbsAddr )`. As is clear from the stack-effect comment, this word transforms an address relative to the base of the active object to an absolute address. For completeness, I added the reverse operation `-↑ ( AbsAddr --`

6. A virtual method, or in Smalltalk terminology a "message," is a piece of code which is subject to late binding. As opposed to a static method, which is unique and defined only in one class so it can be identified at compile time, a virtual method means a series of subroutines—one for each member of a family of classes—thus, the actual routine can be chosen only at run time when the active object is known.
7. Besides the theoretical considerations mentioned above, this approach has an implementation advantage. The object pointer can be very easily implemented by dedicating a CPU register for it. (In my implementation, I used two registers: one for keeping the address of the object and one for keeping the address of its Virtual Method Table—the VMT.) In this way, the cost of the field accesses and method calls can be greatly reduced.
8. Consider a large database which consists of a great many objects interconnected via pointers. When the database image is saved to disk and it has to be reloaded to a different address, all the pointers will become invalid and must be fixed. On the other hand, if the pointers are relative they will remain valid no matter what the physical address of the database is.

Figure Two.

```
1 Class ClassA ClassA DEFINITIONS
Method M1 ( ???? )    ( The stack effect should be recorded here. )
As M1 use: <....> ;M ( Some action. )
1 Class ClassB ClassB DEFINITIONS
Method M2 ( ???? )
As M1 use: <.....> ;M
As M2 use: ClassA M1 ( Call the method M1 of the ClassA. )
           ClassB M1 ( Call the M1 method of the active object. )
                  ;M
```

RelAddr) (a better notation would be ↓, but the latter arrow is not included in the common character sets). Using the ↑ notation, the fields of an object can be represented as offsets from the base address of the object. This is the less-sophisticated way of accessing data in an object. In order to find out more about the behaviour of the objects, we must take a look at their classes. While an object instance is an individual chunk of data with a set of methods to manipulate it, a class is a set of objects which share the same set of methods.

Vocabulary and Class Hierarchy

In order to implement OOP, I chose an old fashioned fig-Forth-style vocabulary structure. I also kept the old system variables CONTEXT and CURRENT to hold the address of the search and definition vocabularies. I have defined the rules of searching so that not only the CONTEXT vocabulary is searched but, if a name cannot be found in the CONTEXT vocabulary, the search goes on with its ancestors.[9]

A class is a special vocabulary with late binding support. This means a class has a VMT[10] which contains the addresses of the virtual methods belonging to the class. There is one class called Objects which is the root of the object hierarchy or, in other words, is the common ancestor of all classes. A new child class can be defined by using the word Class (NewMethods --). For example, the following line will define a new class, *ClassA:*

```
4 Class ClassA
```

The header of *ClassA* will contain a VMT with four more entries than the parent class of *ClassA*. The VMT of the parent class will be copied to the child's VMT, thus the virtual methods will be inherited by default.

Methods

New method names can be assigned to the new entries in the VMT by using the definition word Method. The line below will define a word NewMethod1 in the CURRENT class (remember that a class is just a special kind of vocabulary):

```
Method
NewMethod1
```

The index of the first undefined entry in the VMT of the CURRENT class will be assigned to the method. Notice that the definition of the name of a new method does not specify the action performed by the method. The latter must be defined later by using the word

```
use: ( MethodIndex -- )
```

The index corresponding to a method's name can be obtained by using the word

```
As ( -- MethodIndex)
```

So the definition of a method's body will look like:

```
As NewMethod1
use: <action to be taken> ;M
```

This even looks like an English sentence, thus this notation makes the source more readable. Any method name visible from the CURRENT class can be used. In this way, not only the new methods can be defined, but the old ones defined in the ancestor classes can also be re-defined.

One further advantage of this approach is that the compilation is entirely incremental. Method names and method bodies can be defined in any order. When a method name appears in a definition, the following rules determine what code will be generated for it:

- If the CONTEXT class is an ancestor of the CURRENT one, a "static" call is generated, which means the binding is done at compilation time. The effect of this behaviour is similar to Turbo Pascal's AnAncestor.AMethod; type of statement. If a method is mentioned, not only by having its name specified but by having its type and name specified together (this makes sense only in a method of a successor type), then that method no longer identifies a series of routines, but only one routine which is known at compile time.
- If CONTEXT and CURRENT are the same or belong to different hierarchy, the emitted code uses late binding, which means that a call is generated to a routine with a certain index in the VMT of the currently active object. Figure Two shows an example.

"Static methods" can also be defined. They are otherwise-ordinary Forth words which operate on the active object (or, in other words, they belong to a certain class of objects which can use them to perform certain tasks). The concept of static methods does not add anything new to the OOP support, but it arises as a side product of this implementation of classes and objects. Nevertheless, static methods can be useful in factoring the virtual methods.

9. The term "ancestor" seems rather intuitive to me but, for those who like definitions, the following will do: Vocabulary A is the parent of vocabulary B if B was created with A being the CURRENT vocabulary. Vocabulary X is an ancestor of vocabulary Y if X is the parent of Y or X is an ancestor of the parent of Y. In other words, B is a child of A and Y is a successor of X.
10. The term VMT, as most of my terminology, is borrowed from Turbo Pascal [2].

Obtaining the Standard Size and the Address of the Virtual Method Table of a Class

Every class has a standard size (stored in the header of the class). Also, every class has a VMT table (as part of the class header). The size of the `CONTEXT` class can be accessed by using the words `SizeOf (-- Addr)` or `[Size] ( -- Addr)`. The address supplied in both cases is the address of the cell containing the standard size of the class. I am talking about "standard size" here because, in some cases, objects belonging to the same class can have different sizes (e.g., arrays). It is the responsibility of the programmer to keep the size information recorded in the class header valid, but some tools to facilitate this are provided in the program.

The address of the VMT table can be obtained by `VMTof ( -- VMT)` or by `[VMT] ( -- VMT)`.[11]

The Memory Layout of an Object

The only link between an object instance and its class is the address of the VMT table. This address is stored in every object in the cell immediately before the base address of the object. Yes, this means that the VMT occupies a *negative* offset. I have found that using a negative offset reduces the possibility of accidental errors when testing objects interactively. It is always tempting, especially during debugging, to access an object as if the latter was an ordinary variable. If the VMT address is stored at a negative offset, objects really become similar to variables and other data structures defined by `CREATE ... DOES>`. Thus, the first usable data field begins at the base address of the object. One method, using the word ↑ for accessing data inside the objects data area, has already been mentioned. Another way of manipulating data inside the object is to use fields. The definition word `Field ( Offset -- Offset+CELL)` can be used for creating fields with meaningful names. The following line will create a word *Year* and leave 12 (in my implementation) on the stack.

```
 8 Field Year
```

When the word *Year* is executed, it will leave the base address of the active object plus 8 on the stack, which is the absolute address of the field of the active object called *Year*. In order to make the declaration of fields even easier, two new words can be introduced:

```
: Fields     ( -- 1st-unused-offset )
  SizeOf @ ;
: End-Fields ( 1st-unused-offset -- )
  SizeOf ! ;
```

By using these new words, the declaration of the new fields of a class will look like the following:

```
Fields
Field F1
Field F2
End-Fields
```

In this way, the first new field of the class begins after the last field of the parent class (the size information is copied together with the VMT when a new class is declared), so that the fields of the parent are inherited. Also, the size of the class is taken care of because `End-Fields` will store the offset of the first unused byte, which is the same as the size of object's data area in bytes.[12]

Constructors

In order to create instances of a given class, one needs definition words. If the objects are allocated on the heap,[13] the address of the VMT table still has to be assigned to it and its fields need to be initialized. Because objects belonging to one class can be located in different areas (e.g., dictionary space and heap), I decided to implement a word which initializes the data area of an existing object. This word is called `Init` and it is implemented as a virtual method.[14] If the objects have individual names and are located in the Forth dictionary, the simplest way to produce them is via definition words. One such definition word might be the following:

```
: Obj ( <list of initial values> -- )
  VMTof CREATE , HERE SizeOf @ ALLOT
  { Init } DOES> ( -- Object) CELL+ ;
```

Notice that the effect is similar to that of `VARIABLE` in older Forth systems where an initial value had to be supplied.

An Example

To demonstrate the capabilities of my Forth system, I created the example in Listing One *[pp. 16–17]*. The base class *Numbers* has some methods—ways of behaviour—common to all numbers. This is an abstract class, so it cannot be *instantiated;* that means an object belonging to the class *Numbers* cannot be created.[15] The two derived classes *Integers* and *Rationals* implement meaningful kinds of numbers. The latter two *can* be instantiated. After compiling the example, we can define different kinds of numbers. The following line will create two rational numbers, *R1* and *R2:*

```
Rationals 60 30 Obj R1 81 3 Obj R2
```

The world *Rationals* does not do anything but change the `CONTEXT` class. In other words, it specifies the type of the new object (`Obj` always uses `CONTEXT` to determine the type of the object to be created). Afterwards, their values

11. The difference between `[Size]` and `SizeOf` (also between `[VMT]` and `VMTof`) is the same as the difference between `[']` and `'`.

12. In machines which are not capable of addressing individual bytes, the indication of the object's size in bytes can be meaningless. In my implementation, it is the easiest way to go because, on the 486 even in 32-bit mode, individual bytes are accessible.

13. My program does not yet have built-in `Memory-Allocation` wordset support but, for the time being, any standard definition of this wordset will do. I have found Gordon Charlton's `ANS HEAP` to be useful.

14. Although `Init` is quite different from the virtual methods: if a virtual method with a certain name is defined in a class with a given stack effect, it is supposed to have the same stack effect in all the successor classes. This is not true for `Init` and it is just an implementation trick to define this word as a "virtual method."

15. In reality, the definition word `Obj` will create such an object, but any attempt to call its methods will trigger an error message.

can be examined easily:

```
R1 . R2 .     1 / 2  1 / 27 Ok
```

(*R1* and *R2* are already normalized). An addition is also simple:

```
R1 R2 + .     29 / 54 Ok
```

In the example, the value of *R1* is changed and it equals 29/54. Objects belonging to the class *Integers* can be treated the same way:

```
Integers 20 Obj I1 50 Obj I2  Ok
I1 . I2 .    20 50 Ok
I1 I2 + .    70 Ok
```

In fact, the same words (., ,, +, -, *, and /) can handle them.[16]

The Accessibility of the Information

Information hiding is one of the usual features of an OOP language. The pioneers of OO-Forth spent a lot of effort on it [3]. On the other hand, Forth is very often used by hardware developers, hackers, and similar people who definitely will ignore such an effort. So I decided not to bother with it. One cannot really "physically seal" a piece of memory from experts. Also, to let the user know what is going on "behind the scenes" saves a lot of trouble during debugging. This does not mean that I want to encourage hacking around the internal parts of an object—which would render my whole effort spent on implementing this OOP support meaningless. I simply do not believe that a programmer who is not willing to use proper techniques can be forced to do so.

On the other hand, the encapsulation is rather good in my model. In principle, nothing from the outside can access the data area of an object; even the address of a field cannot be calculated. Only the object itself can "make it known" to the rest of the application.

One advantage of Forth is that words are not split into categories as in other languages. There are no such things as "operators," "keywords," or "identifiers." This lack of differentiation means the programmer has more freedom to change the underlying implementation, provided that the stack effect (thus, the interface to the rest of the application) remains the same.

Because of the latter property of Forth, and because of the good encapsulation, the programmer can hide the implementation details from the user in my object-oriented Forth dialect, even though the information would not be "physically sealed." To prevent accidental usage of words which are supposed to be "factors" or auxiliary words, one might create aliases of the usable methods in other vocabularies, where the rest of the application's routines reside.

16. At first sight, this "sharing the operators" might look close to the idea of C++'s operator overload, but the latter implies a typed language. In our example, these "operators" can work on any derived class of *Numbers*, but always on two operands belonging to the *same class*, while a C++-style operator should be able to handle such cases as multiplication of a rational number by an integer.

Conclusions

In this paper, a dialect of Forth featuring OOP support and native code generation has been introduced. In this dialect, the code generation—especially the definition of compile-time behaviours—is controlled by the user. In this way, one can decide which parts of the code are important and to be in-lined or substituted by more efficient pieces of machine code, and which are to be left alone. The main feature of the OOP support in this Forth is simplicity. It uses a vocabulary structure, with some extras to implement classes with inheritance and late binding (thus, polymorphism).

Although most of the OOP support words do elementary manipulations (but what do you expect from an assembly programmer?), they can be used for building higher-level interfaces tailored to individual taste and needs. The necessity of a good OO-Forth is obvious these days, but the Forth community still does not have a standard. Although many object-oriented Forth implementations are available, I found my dialect a very convenient one. The program size is small (the kernel is about 32K—32-bit machine code, not threaded code) and the functionality is easy to understand. Because of the nano-compiler approach, a programmer can easily keep in mind what is going on behind the scenes; thus, he/she has better control over the system. The independent and "animate" object instances provide better encapsulation, thus facilitating an even more structured programming style than the usual OO-Forth dialects or C++ and Turbo Pascal.

References

[1] K.D. Veil and P.J. Walker. "Forth Nano-Compilers," *Forth Dimensions*, Vol. XVI No. 2, July-August 1994, 34–37.

[2] Borland International, Inc., 1992. *Borland Pascal With Objects User's Guide.*

[3] Dick Pountain. *Object-Oriented Forth*. Academic Press Ltd., London, 1987.

Acknowledgments

I want to say thanks to the Hung Hing Ying Physical Sciences Research Fund for providing a grant to pay for my 486.

Code: Mr. Zsótér's OOF is available via ftp from taygeta.oc.nps.navy.mil in the /pub/Forth/Reviewed subdirectory. The package consists of two files: (1) OOF.ZIP (about 170 Kb) includes the sources of the VMI protected mode monitor and the OOF 32-bit, object-oriented Forth; all assembly (TASM format) sources and the Forth source files; and the .EXE programs. (2) OOFLST.ZIP contains *only* the .LST files generated by the assembler (about 240 Kb).

The program runs on '486 and '386 machines. The package is under GNU General Public License Version 2. The present version is labelled 0.8 to indicate that changes are needed (although it seems stable, it is not "bug free").

András Zsótér first read about Forth in a magazine. Working in a hospital ("...we had very old machines there—some were older than me"), he programmed the then-new ZX-Spectrums in assembly to perform signal acquisition from EEGs. At Jozsef Attila University in Szeged, Hungary (where he degreed in Chemistry and General & Applied Linguistics), he started to learn Forth. He had learned Pascal in an afternoon, but it took him a month to get a feel for Forth. He discovered that practicing Forth improved his programming style in other languages, so he kept experimenting with it.

Since January 1, 1993, he has been in a Ph.D. program in the Department of Chemistry at Hong Kong University, where he synthesizes organic compounds via robotics. When he needed a programming tool, Forth was *the* natural solution, so he implemented one in the way described in this paper.

Listing One. Examples demonstrating the system's capabilities.

```
FORTH DEFINITIONS

: GCD ( U1 U2 -- GreatestCommonDivisior )
  BEGIN
  2DUP <> WHILE                 ( If U1=U2 either will do. )
   2DUP MIN >R MAX R - R>       ( Substract the smaller from the greater )
  REPEAT                        ( Chech again if U1=U2. )
  DROP ;                        ( One of them is enough. )

Objects DEFINITIONS
5 Class Numbers Numbers DEFINITIONS
Method Add ( Number -- )
Method Sub ( Number -- )
Method Mul ( Number -- )
Method Div ( Number -- )
Method Print

0 Class Integers Integers DEFINITIONS
Fields
Field N
End-Fields

As Init use: ( N -- ) N ! ;M

As Add use: { N @ } N +! ;M

As Sub use: { N @ } NEGATE N +! ;M

As Mul use: { N @ } N @ * N ! ;M

As Div use: { N @ } N @ SWAP / N ! ;M

As Print use: N @ . ;M

Numbers DEFINITIONS
2 Class Rationals Rationals DEFINITIONS
Fields
Field NUM
Field DEN
End-Fields
Method Normalize ( -- )
Method Invert    ( -- ) ( 1/x )
\end{verbatim}
\pagebreak
\begin{verbatim}
As Init use: ( DEN NUM -- ) NUM ! DEN ! Normalize ;M

: (Add) ( num den -- )            ( A "factor" of Add )
        ( This is not the best way to add two rational )
        ( numbers but as an example it will do.        )
   DUP NUM @ * NUM !              ( NUM*den                 )
   DEN @ SWAP OVER * DEN !        ( DEN*den => DEN          )
   * NUM +!                       ( num*DEN+NUM*den => NUM )
   Normalize ;

As Add use: { NUM @ DEN @ } (ADD) ;M

As Sub use: { NUM @ NEGATE DEN @ } (ADD) ;M

As Mul use:
    { NUM @ DEN @ }       ( Get the values of the other Rational. )
    DEN @ * DEN !         ( Multiply denominator by the other's denominator.)
    NUM @ * NUM !         ( Multiply numerator by the other's numerator.)
    Normalize ;M

As Div use:
    { NUM @ DEN @ }       ( Get the values of the other Rational. )
    NUM @ * SWAP          ( Multiply numerator by the other's denominator.)
    DEN @ * NUM ! DEN !   ( Multiply denominator by the other's numerator.)
    Normalize ;M

As Print use: NUM @ . ." / " DEN @ . ;M

As Normalize use:
    DEN @ NUM @             ( Get denominator and numerator. )
    2DUP XOR >R             ( A not quite ANSI way to determine the sign. )
    ABS SWAP ABS            ( Calculate the absolute value of both. )
    2DUP GCD                ( Calculate the GCD. )
    DUP >R / DEN !          ( Normalize the denominator. )
    R> /                    ( Normalize the numerator. )
    R> 0< IF NEGATE ENDIF   ( Adjust the sign. )
    NUM ! ;M

As Invert use: NUM @ DEN @ NUM ! DEN ! Normalize ;M
\end{verbatim}
\pagebreak
\begin{verbatim}
Numbers DEFINITIONS

( An now some words that look useful to an ordinary Forth programmer. )

: . ( Number -- ) { Print } ;

: + ( Number1 Number2 -- Number1+Number2 ) SWAP { Add O@ } ;

: - ( Number1 Number2 -- Number1-Number2 ) SWAP { Sub O@ } ;

: * ( Number1 Number2 -- Number1*Number2 ) SWAP { Mul O@ } ;

: / ( Number1 Number2 -- Number1/Number2 ) SWAP { Div O@ } ;
```

MAKE YOUR SMALL COMPUTER

THINK BIG

(We've been doing it since 1977 for IBM PC, XT, AT, PS2, and TRS-80 models 1, 3, 4 & 4P.)

FOR THE OFFICE — Simplify and speed your work with our outstanding word processing, database handlers, and general ledger software. They are easy to use, powerful, with executive-look print-outs, reasonable site license costs and comfortable, reliable support. Ralph K. Andrist, author/historian, says: "FORTHWRITE lets me concentrate on my manuscript, not the computer." Stewart Johnson, Boston Mailing Co., says: "We use DATAHANDLER-PLUS because it's the best we've seen."

MMSFORTH System Disk	from $179.95

Modular pricing — Integrate with System Disk only what you need:

FORTHWRITE - Wordprocessor	$99.95
DATAHANDLER - Database	$59.95
DATAHANDLER-PLUS - Database	$99.95
FORTHCOM - for Communications	$49.95
GENERAL LEDGER - Accounting System	$250.00

FOR PROGRAMMERS — Build programs FASTER and SMALLER with our "intelligent" MMSFORTH System and applications modules, plus the famous MMSFORTH continuing support. Most modules include source code. Ferren MacIntyre, oceanographer, says: "Forth is the language that microcomputers were invented to run."

SOFTWARE MANUFACTURERS — Efficient software tools save time and money. MMSFORTH's flexibility, compactness and speed have resulted in better products in less time for a wide range of software developers including Ashton-Tate, Excalibur Technologies, Lindbergh Systems, Lockheed Missile and Space Division, and NASA-Goddard.

MMSFORTH V2.4 System Disk	from $179.95

Needs only 24K RAM compared to 100K for BASIC, C, Pascal and others. Convert your computer into a Forth virtual machine with sophisticated Forth editor and related tools. This can result in 4 to 10 times greater productivity.

Modular pricing — Integrate with System Disk only what you need:

EXPERT-2 - Expert System Development	$69.95
FORTHCOM - Flexible data transfer	$49.95
UTILITIES - Graphics, 8087 support and other facilities.	

and a little more!

THIRTY-DAY FREE OFFER — Free MMSFORTH GAMES DISK worth $39.95, with purchase of MMSFORTH System. CRYPTOQUOTE HELPER, OTHELLO, BREAKFORTH and others.

MMSFORTH

MILLER MICROCOMPUTER SERVICES
61 Lake Shore Road, Natick, MA 01760
(508/653-6136, 9 am - 9 pm)

Call for free brochure, technical info or pricing details.

Forth Scientific Library Project

Everett "Skip" Carter
Monterey, California

[Following is a report on the status of the Forth Scientific Library Project as of January 3, 1995. Regular updates may be found on comp.lang.forth or by contacting the author directly (see information at end of article). —Ed.]

Participants

Warren Bean	Charles G.Montgomery
Richard Beldyk	Leonard Morgenstern
Gary Bergstrom	Julian Noble
Jim Brakefield	Fabrice Pardo
Gus Calabrese	Michel W. Pelletier
Skip Carter	Penio Penev
ChihYu Jesse Chao	Elizabeth Rather
Gordon Charlton	Tony Reid-Anderson
Munroe C. Clayton	Richard Rothwell
Glen Haydon	Stephen Sjolander
Marcel Hendrix	John Svae
Chris McCormack	Andrejs Vanags

(Mail to scilib@taygeta.oc.nps.navy.mil will be automatically distributed to all the participants listed above.)

Code Contributions

Contributed but not reviewed:

Quasi-Random number generation	Skip Carter
Monte Carlo Row inverse (ACM 166) and related algorithms	Skip Carter
Telescope 1 (ACM 37) (reduction of degree of polynomial approximations)	Skip Carter
Telescope 2 (ACM 38)	Skip Carter
Coefficient Determination (ratio of polynomials) (ACM# 131)	Skip Carter
Reversion of Series (ACM # 193)	Skip Carter
Weibull PDF and Weibull Random variables	Skip Carter
Linear and Circular (discrete) Convolution	Skip Carter
Complex math operations (magnitude, power, multiplication and division)	Skip Carter
Polynomial transformer (ACM #29)	Skip Carter
Jacobian elliptic functions	Skip Carter
Nonlinear transformation of series (SHANKS) (ACM #215)	Skip Carter
Finite segment of Hilbert Matrices, their inverses and determinants	Skip Carter
LU Factorization of square matrices	Skip Carter
Back-substitution solution for LU factored linear systems	Skip Carter
Solution of linear Fredholm equation of the second kind	Skip Carter
Solution of a set of Volterra equations of the second kind	Skip Carter
Inverse of an LU factored matrix	Skip Carter
Determinant of an LU factored matrix	Skip Carter
Square root of a square symmetric matrix	Skip Carter
Adjustment of Matrix inverse when an element is perturbed (ACM #51)	Skip Carter
Eigenvalues and Eigenvectors of a real symmetric matrix	Skip Carter
Basic arithmetic and conversions for rational numbers	Gordon Charlton
Permutations and Combinations	Gordon Charlton
16-bit Cyclic Redundancy Checksums	Gordon Charlton
Gauss-Seidel iteration solution to linear systems	Skip Carter
Gauss probability function	Skip Carter
Solution of banded linear systems	Skip Carter
Tools to use polynomial interpolation with a large table	Marcel Hendrix
Basic statistics of a floating point array	Skip Carter
4th order Runge-Kutta solver for systems of ODEs	Skip Carter
FIND nth element of an unsorted array (ACM #65)	Skip Carter
Simulated Annealing using Cauchy cooling	Skip Carter

Currently being reviewed:

Rootfinder (ACM #2)	Skip Carter
Stochastic Differential Equation solver (scalar version)	Skip Carter
Box-Muller transformation (polar form)	Skip Carter
Quadratic Equation solver	Skip Carter
Fast Walsh Transform	Skip Carter
Four methods for Direct Fourier Transforms	Skip Carter
Radix-2 Fast Fourier Transform routines (five versions one, two, and three butterflies, tabular, non-tabular)	Skip Carter
Complete Elliptic Integral of the first kind (ACM #55)	Skip Carter
Complete Elliptic Integral of the second kind (ACM #56)	Skip Carter
Complete Elliptic Integrals of 1st and 2nd kinds (ACM #165)	Skip Carter
Tridiagonal solver, using the Thomas algorithm	Skip Carter
Gauss-Legendre Integration	Skip Carter
First derivative of a function by Richardson extrapolation	Skip Carter
RAN4 Pseudo-random number generator	Gordon Charlton
Regular spherical Bessel functions jn(x), n=0-9	Julian Noble

Reviewed:
(Reviewed code is available via anonymous FTP at taygeta.oc.nps.navy.mil/pub/Forth/Scientific

or via WWW at
http://taygeta.oc.nps.navy.mil/scilib.html)

1. Real Exponential Integral (ACM #20)	Skip Carter
2. Complete Elliptic Integral (ACM #149)	Skip Carter
3. Polynomial evaluation by the Horner method	Skip Carter
4. Logistic function and its first derivative	Skip Carter
5. Cube root of real number by Newton's method	Julian Noble
6. Solution of cubic equations with real coefficients	Julian Noble
7. Regula Falsi root finder	Julian Noble
8. Fast Hartley (Bracewell) Transform (with supplemental utilities and tests by Marcel Hendrix)	Skip Carter
9. Aitken Interpolation (ACM #70)	Skip Carter
10.Hermite Interpolation (ACM #211)	Skip Carter
11.Lagrange Interpolation (ACM #210)	Skip Carter
12.Forward and Backward divided differences	Skip Carter
13.Newton Interpolation with Divided differences (ACM 168 & 169)	Skip Carter
14.Factorial	Skip Carter
15.Shell sort for floating point arrays	Charles Montgomery
16.Exponentiation of a series (ACM # 158)	Skip Carter
17.Polynomial and Rational function interpolation and extrapolation	Marcel Hendrix
18.The Gamma, LogGamma and reciprocal Gamma functions	Skip Carter
19.Adaptive Integration using Trapezoid rule	Julian Noble
20.Parabolic Cylinder functions and related Confluent Hypergeometric functions	Skip Carter
21.Special Polynomial (Chebyshev, Hermite, Laguerre, Generalized Laguerre, Legendre, and Bessel) Evaluation	Skip Carter
22.Conversion between calendar date and Julian day (ACM 199)	Skip Carter
23.R250 (also minimal standard) Pseudo-random number generator	Skip Carter

Walnut Creek has asked to include the FSL on their *Algorithms* CD-ROM.

Participation by all those interested in using Forth for scientific applications is welcomed. Contribute whatever you feel comfortable working with. Contributors will get a free copy of the Walnut Creek CD-ROM when it becomes available.

You can join using a WWW form (follow the links from my home page) or by sending me e-mail; a mail-server service is in the works as well.

Dr. Everett "Skip" Carter is an Assistant Professor of Oceanography at the Naval Postgraduate School. He wrote the Forth system for, and helped design, the RAFOS float which is being used internationally as part of the World Ocean Circulation Experiment. He can be reached at the following:

Internet:	skip@taygeta.oc.nps.navy.mil
UUCP:	...!uunet!taygeta!skip
WWW:	http://taygeta.oc.nps.navy.mil/skips_home.html

Total control with *LMI FORTH*™

For Programming Professionals: an expanding family of compatible, high-performance, compilers for microcomputers

For Development:
Interactive Forth-83 Interpreter/Compilers for MS-DOS, 80386 32-bit protected mode, and Microsoft Windows™
- Editor and assembler included
- Uses standard operating system files
- 500 page manual written in plain English
- Support for graphics, floating point, native code generation

For Applications: Forth-83 Metacompiler
- Unique table-driven multi-pass Forth compiler
- Compiles compact ROMable or disk-based applications
- Excellent error handling
- Produces headerless code, compiles from intermediate states, and performs conditional compilation
- Cross-compiles to 8080, Z-80, 64180, 680X0 family, 80X86 family, 80X96/97 family, 8051/31 family, 6303, 6809, 68HC11
- No license fee or royalty for compiled applications

Laboratory Microsystems Incorporated
Post Office Box 10430, Marina Del Rey, CA 90295
Phone Credit Card Orders to: (310) 306-7412
Fax: (310) 301-0761

Offete Enterprises, Inc.
1306 South B Street
San Mateo, California 94402
Tel: (415) 574-8250 Fax: (415) 571-5004

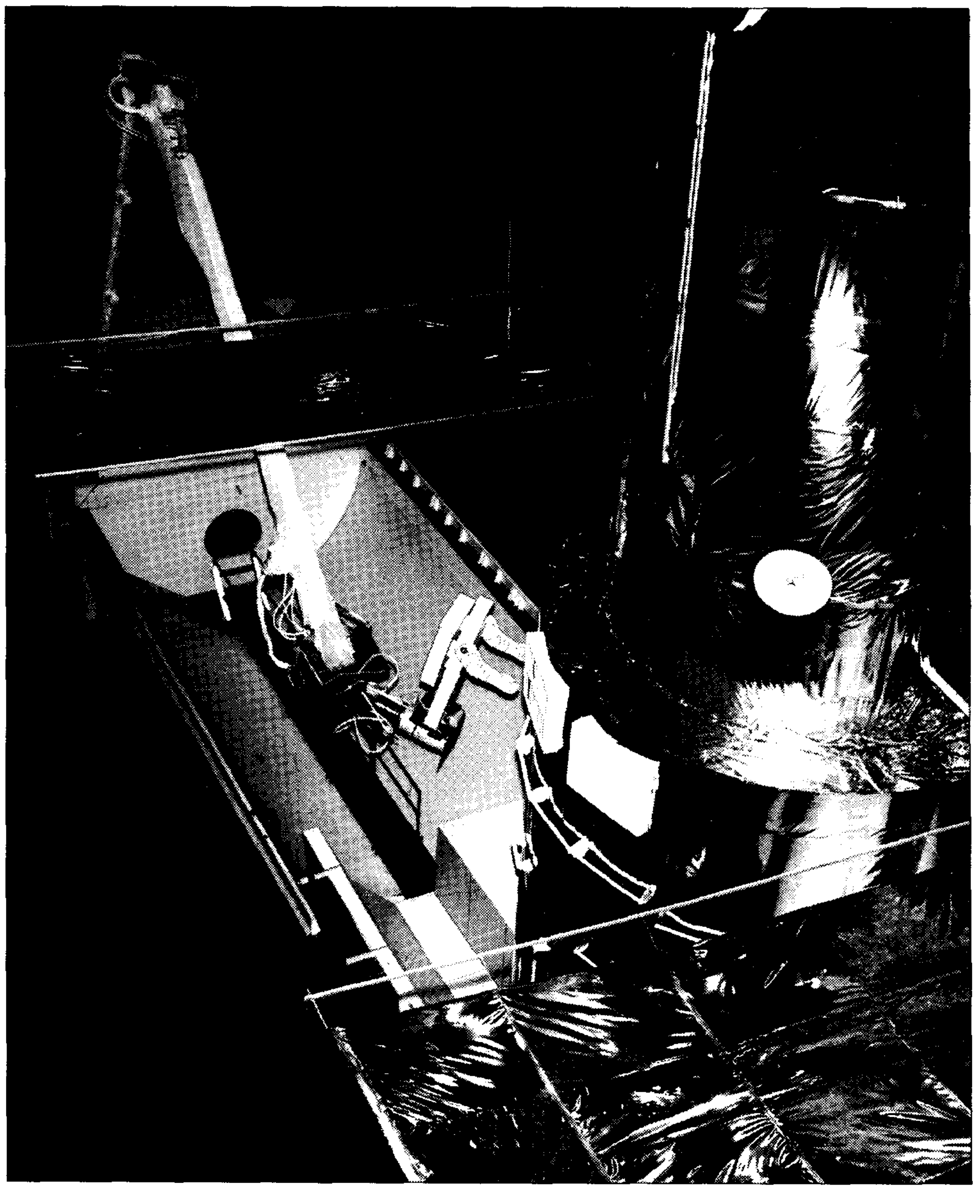
MuP21 Products

1. **MuP21 Chip** designed by Chuck Moore, $25
   80 MIPS CPU with Video Coprocessor
2. **MuP21 Evaluation Kit**, $100
   MuP21, ROM, PCB and software
3. **Assembled MuP21 Evaluation Kit**, $350
   Above Kit assembled with 1Mx20 DRAM
4. **MuP21 Programming Manual**,$15.00
5. **MuP21 Advanced Assembler**
   by Robert Patten, $50
6. **MuP21 eForth** by Jeff Fox, $50
7. **More on Forth Engines**
   Volume 18, June 1994 — $20.

U.S. bank draft, money order accepted
Add 10% (up to $10) for air shipping
Californians please add 8.25% sales tax

A Simulator for NASA's Shuttle Robot Arm

Edward K. Conklin
Manhattan Beach, California


—Photo courtesy of NASA. Enhanced and retouched by Remote Control.

NASA's space shuttle carries a 50-foot long, six-joint arm for use in satellite deployment and retrieval operations, and to assist astronauts in servicing tasks such as the recent mission to repair and upgrade the Hubble Space Telescope. The arm, formally called the Remote Manipulator System (RMS), has ten different modes of operation, ranging from simple direct movement of the joints, one at a time, to very complex multi-joint motions directed by rotational and translational joysticks. Using the joysticks, an operator can command motion about any desired axis, and the RMS software will make all the coordinate transformations and complex calculations necessary to derive the needed command rates for each of the six joints. Arm control, status information, and positional displays are provided both by a hard-wired panel containing control switches, status lights, and digital displays, and by a series of interactive status and control screens on the shuttle's General Purpose Computer (GPC).

In order to plan for missions involving the RMS, there are also two ground-based versions of it, one at the Johnson Spaceflight Center in Texas (JSC) and one at the Goddard Spaceflight Center in Maryland (GSFC). Because of the need to work in a gravity environment and other specific design factors, the ground-based arms differ from the RMS and from each other. The GSFC arm, for example, is designed to carry up to a thousand-pound payload at its tip. In order to do this, it uses a high pressure (4000 psi) hydraulic system rather than electric motors as on the RMS.

In June 1994, Forth, Inc. was selected to provide the overall control program for the GSFC arm, called the RMSS

(Remote Manipulator System Simulator), along with Electrologic of America (ELA) who provided the control electronics and drivers. Because of the completely different nature of the joint controls, the original RMS software was not usable except as a source of algorithms. The basic requirements for the RMSS were that it must behave identically to the RMS as far as operational modes, display panels, and CRT screens were concerned, while interfacing to a new and different type of hydraulic control hardware. There were other constraints, such as the fact that, although this is a rate control system, the RMSS (unlike the shuttle RMS) has no tachometers and rate information had to be derived from differencing angular position data. Finally, the entire system was to be delivered in 60 days.

The RMSS proved to be an excellent application for EXPRESS, Forth, Inc.'s Event Management and Control System software package. The design methodology in EXPRESS, involving the factoring of the application into separate, largely autonomous modules called "processes," was an important factor in keeping the development time—and particularly the debugging time—short. The RMSS contains fourteen separate processes: one for each joint, one for each joystick, one for the digital display panel, a simulation process, a trending process, and several supervisory processes. Each process was developed and tested as a stand-alone unit, and was later integrated into the complete system.

EXPRESS contains, as a standard feature, full simulation capabilities which allow application testing without any I/O hardware present. During RMSS development, engineers simulated operation of the arm in all modes, including the joystick-commanded multi-joint motions with their involved mathematics. The simulation process in the RMSS application was used to convert commanded rates to simulated joint angles so that the full arm position

...when the software was installed, not a single change was made to the executive control algorithms.

was available on all the system displays. Simulation testing was so thorough that when the arm software was installed on site, not a single change was made to the executive control algorithms.

When working with the real arm hydraulic system, it became clear that major changes were needed in the low-level joint control processes to ensure smooth and accurate operation. First, because joint angle readouts are used for both position and rate, precise calibrations were necessary. EXPRESS' standard Historical Trending Display (HTD) was used to capture joint angles as a function of time; then later analysis of the graphs provided the necessary position and rate information.

Because of the large and changeable gravity loads on the system, it was not possible to achieve precise commanded velocities for the various joints without rate feedback. A PID control loop was developed for each joint, with rate feedback coming from an adjustable second-order software filter on the joint angle. This complexity was necessary because differencing angles to get rates is inherently a noisy process. Although this was a major change to each joint process in the RMSS, because of the inherent process isolation in EXPRESS there were no system-wide ramifications. In order to tune the filter coefficients and PID constants for each joint, EXPRESS' Process Monitor Display (PMD) was used extensively. This utility allows examination and on-line modification of all variables in a process. While the arm was running, it was possible to change the characteristics of each joint and observe the results without stopping to recompile and reload the software. In a single operating session of a few hours it was, therefore, possible to make a complete pass at optimizing all six joints in the system.

The complete checkout, debugging, and optimization of the RMSS on site took several weeks, including the inevitable hardware component failures and software modifications typical of first-time operation of a complex system. At the end of this period, in a two-day demonstration the RMSS operation successfully duplicated the original shuttle RMS. Astronauts from NASA's Johnson Spaceflight Center who had been trained on the RMS were able to use the RMSS after only a few minutes of explanation of the essential differences.

The author, who generally goes by "Ned," was one of the founders of Forth, Inc. In his former life, he was a radio astronomer at the National Radio Astronomy Observatory and NAIC, Arecibo, Puerto Rico.

Ruffer
Graphic Solutions

Computer Graphics
24-Bit Color Scanning
Superimposing
Special Effects

Internet: druffers@ix.netcom.com
Voice: 1(310) 372-0671 M–F 9am–5pm
Snail: P.O. Box 642
Hermosa Beach, CA 90254–0642

Vehicular Rollover in Accident Reconstruction

J.V. Noble
Charlottesville, Virginia

Abstract

This paper present a numerical simulation of vehicular rollover accidents on both wet and dry pavements, with graphical display of flying cars. One of the more unusual features is the use of complex arithmetic to describe rigid-body motion in two dimensions.

We hope the information contained in this note proves useful in planning your next demolition derby.

Introduction

My acquaintance with Forth began in response to a need to accelerate calculations of vehicular accident simulations and reconstructions, for use in litigation.[1]

Recently I have been studying vehicular rollover arising from sideward skidding into a curb or other obstacle. A motor vehicle sliding sideways on a pavement can roll over as a result of collision with a barrier—such as a curb—that "pins" the wheels. The behavior of a car—idealized as a rigid body—under these conditions can be quite complex.

The descriptions of such one-car accidents using theoretical mechanics becomes fairly involved even when we restrict the motions to two dimensions rather than three. We cannot use the Lagrangian methods we study in advanced mechanics courses because the constraints in the problem are non-holonomic.[2] Second, we must include friction, a non-conservative force. We therefore fall back on Newton's Second Law of Motion,

$$\vec{F} = m\frac{d^2\vec{x}}{dt^2} \qquad (1)$$

and its rotational analogue,

$$\vec{N} = I\frac{d^2\vec{\theta}}{dt^2} \qquad (2)$$

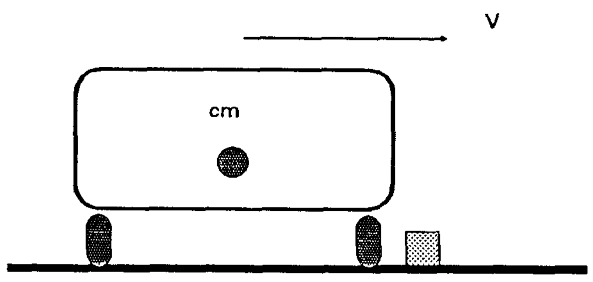
The total force $\vec{F}$ and torque $\vec{N}$ derive from the forces of the tires against the pavement—these arise in turn from friction and the compression of the tire by the vehicle's weight. When the tires collide with the curb, the forces increase drastically and must be modelled carefully.

In the only previous study known to me,[3] the author simplified the problem in two ways:

- He treated the forces as impulsive, that is, very large in magnitude and of short duration, so they could be regarded as changes of momentum;
- He confined the motion to the x-y plane, permitting only rotations about the car's longitudinal axis (z-direction), perpendicular to the plane of the center-of-mass motion.

The impulsive approximation to the tire-curb collision works like this: taking the car's cm to be $\frac{h}{2}$ from the ground, and $\frac{w}{2}$ from either side of the car (Figure One, below), and assuming both momentum and angular momentum conservation during the (brief) time of impact, the car acquires new linear and angular velocities $\dot{X}_f$, $\dot{Y}_f$, $\dot{\theta}_f$, immediately following the collision. Following the impact there are no torques and only the force of gravity acts on the car as it flies through the air.

Figure One. Car sliding into a barrier.



1. Accident Analysis Associates, Inc. reconstructs accidents, advises attorneys, and provides expert testimony for vehicular and other accidents.
2. See, e.g., H. Goldstein, *Classical Mechanics*, 2nd ed. (Addison-Wesley Publishing Co., Reading, MA, 1980).
3. Ian S. Jones, *The mechanics of rollover as the result of curb impact*, SAE paper #750461 (1975).

The equations of momentum and angular conservation are

$$M\dot{X}_f - MV = \Delta P_x \equiv \int F_x\,dt \qquad (3x)$$

$$M\dot{Y}_f = \Delta P_y \equiv \int F_y\,dt \qquad (3y)$$

$$\int N_z\,dt \equiv \frac{w}{2}\Delta P_y + \frac{b}{2}\Delta P_x = I\,\dot{\theta}_f. \qquad (4)$$

We need to determine five unknowns, impossible with only three equations. So we must apply some other (approximate) condition. We see that the wheel that collides with the curb does not have any velocity component in the y direction (at first). So we can say

$$0 = \dot{Y}_f + \frac{w}{2}\dot{\theta}_f\,. \qquad (5)$$

However, what about the instantaneous x-component of velocity acquired by the near wheel immediately after the collision? Here there are two extreme cases: in the first, the collision is elastic, so the wheel has x-velocity $-V$; in the second, the collision is completely inelastic, hence the wheel has x-velocity 0. In general, then,

$$\dot{X}_f + \frac{b}{2}\dot{\theta}_f = -cV \qquad (6)$$

where c is the "coefficient of restitution."

Equations 3x,y and 4 are now easy to solve explicitly using conditions 5 and 6. We get an equation in $\dot{\theta}_f$ which yields

$$\dot{\theta}_f = -\frac{1+c}{2}\,\frac{bMV}{I+M\,r^2}\,, \qquad (7\theta)$$

$$\dot{X}_f = \left((1+c)\,\frac{M\,b^2/4}{I+M\,r^2} - c\right)V \qquad (7x)$$

$$\dot{Y}_f = (1+c)\,\frac{M\,bw/4}{I+M\,r^2}\,V \qquad (7y)$$

where the distance r from the center of mass to the wheel is

$$r^2 = \left(\frac{b}{2}\right)^2 + \left(\frac{w}{2}\right)^2.$$

If we calculate the kinetic energy before and after, using the formula

$$E_f = \frac{1}{2}\,I\dot{\theta}_f^2 + \frac{1}{2}M\dot{X}_f^2 + \frac{1}{2}M\dot{Y}_f^2 \qquad (8)$$

we find that, except in the elastic case, energy is not conserved. With:

$$E_i = \frac{1}{2}MV^2,$$

we find that

$$E_f = E_i\left[c^2 + \frac{M\,b^2\,(1-c^2)/4}{I+M\,r^2}\right]. \qquad (9)$$

Perhaps the most interesting prediction of the fully elastic case is that the car will bounce backward from the barrier, i.e., $\dot{X}_f$ is negative!

The major shortcoming of Jones' impulsive treatment arises from his treatment of the friction between tire and curb. There is some vertical motion while the tire rubs on the curb, and consequently a frictional force that opposes it. There is some uncertainty as to the direction and magnitude of the frictional force, since the instantaneous velocity of the point of contact between the tire and curb is not well defined.

Motivation for a More-Detailed Model

To do better than the impulse approximation, we must model the actual behavior of the forces with time. That is, we must model how the tires respond to static and dynamic loads, their instantaneous velocities at their points of contact with pavement or curb, and the energy absorbed in the collision with the curb.

Part of the motivation for proceeding this way is the folklore that cars turn over more readily on dry pavement than on wet. By modelling the forces in detail one may hope to understand the accuracy of the assumptions made in the impulse approximation, as well as to obtain more-detailed comparison with experiment.

The unfortunate paucity of disposable cars, combined with a regrettable absence of scientific curiosity—to the point of downright pusillanimity—on the part of friends, relatives, and colleagues of the author of the present article, preclude the presentation of new experimental findings at this time, however.

Details of the Calculation

The calculations reported here were also based on two-dimensional motion in the x-y plane, with the z-axis oriented out of the plane. The major improvement over Jones' earlier work is the direct integration of the equations of motion. For this we need detailed force laws. We assume that the tires are Hooke's Law compression springs that provide a force proportional to the depth of compression and in the opposite direction. That is, the force law of the pavement against the tire is

$$f(y) = \begin{cases} 0\,, & y > a \\ k(a-y)\,, & y \le a \end{cases} \qquad (10)$$

A similar law acts to the left when the near tire impinges on the curb. The force constant is adjusted so that when the car is simply sitting on the pavement, the deflection is $\frac{a}{2}$ i.e., the force constant is given by

$$2\cdot k\cdot\frac{a}{2} = Mg \qquad (11)$$

where g is the gravitational acceleration at the Earth's surface, 9.8 m/s^2, and M is the mass of the car. (That is, Mg is the weight.)

We can deal with energy-absorbing collisions either by including dashpots (shock absorbers) that are described by Stokes' Law, with their resistance proportional to the velocity and opposing it in direction; or by reducing the spring constant k during the expansion subsequent to compression. The new spring constant is related to the coefficient of restitution, c, by

$$k' = k\,c^2.$$

In addition to the tire force (on the car) opposite to the direction of compression, the tire exerts a frictional force tangential to the surface it is sliding on. This force is directed opposite to the (sliding) velocity of the point of contact. The frictional force is conventionally modelled as proportional to the normal force,

$$F_{fric} = -\mu\, F_{normal}\ \mathrm{sign}(\,v_{tangential}\,). \qquad (12)$$

Finally, the equations of motion are simply Newton's laws:

$$\ddot{Z} = \frac{1}{M} F_{tot} \qquad (13)$$

$$\ddot{\theta} = \frac{1}{I} N \equiv \frac{1}{I} \sum_k \mathrm{Im}\left(z_k{}^* F_k\right) \qquad (14)$$

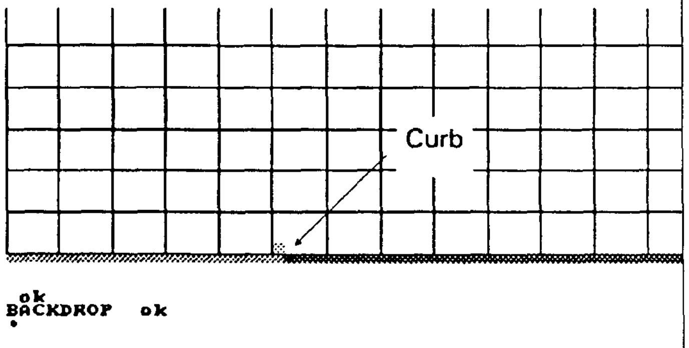
Note that in writing (13) and (14) we are using complex notation. This is a big help because the vector product $\vec{r} \times \vec{F}$ in two dimensions can be expressed as a complex multiplication, as in Eq. (14). Moreover, the rotation matrix applied to a vector can also be expressed as simple complex multiplication (in two dimensions!):

$$Rot(\,\vec{r},\ \theta\,) \equiv e^{i\theta}\, z \qquad (15)$$

Results

The reader might like to see some actual results before we discuss how to solve these equations using Forth. We plot the center-of-mass position of the car, with its orientation at each time "snapshot" drawn as a simple rectangle—fancier graphics seemed like overkill.

The action takes place against a backdrop consisting of the pavement, curb, and a grid of one-meter squares that provides a scale. The actual pictures were captured using



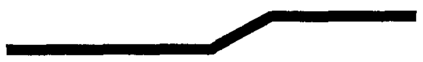
```
ok
BACKDROP  ok
```

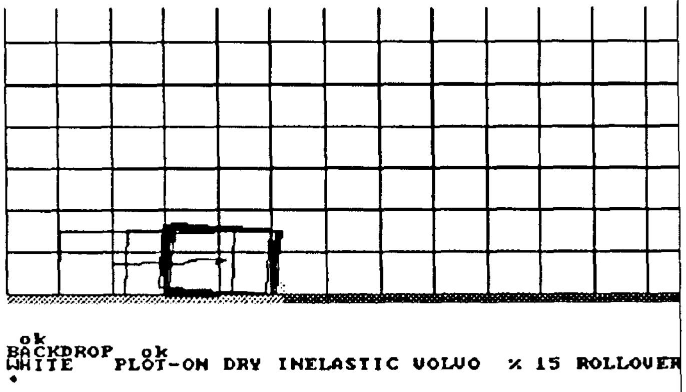
Hijaak directly from the VGA display and were converted to GEM *.img format for printing with this article. The command line makes the figures almost self-explanatory: a run is initiated by saying BACKDROP; then choosing a color for the car (in all these cases, WHITE—bright white on the screen); setting the output mode to plot via PLOT-ON; choosing the friction conditions (so far, WET or DRY); choosing the coefficient of restitution (ELASTIC ⇒ c=1, INELASTIC ⇒ c=0); choosing the vehicle parameters (VOLVO); the initial speed (in miles per hour, e.g., % 20) and then invoking the program, saying ROLLOVER. Needless to say, despite the latter name, the vehicle does not always flip.

The situation investigated here—a vehicle "stubbing its toe" against a fairly high vertical curb—does not bear out the lore that rollovers occur more easily in dry than in wet conditions. We can see this in comparing the 15 and 20 mph (inelastic) collisions, in wet and dry conditions. For either initial speed, the gyrations performed are far more enthusiastic in the absence of friction. The chief reason for this is that the force between tire and curb is very large. Friction then leads to considerable energy dissipation at this point of contact.

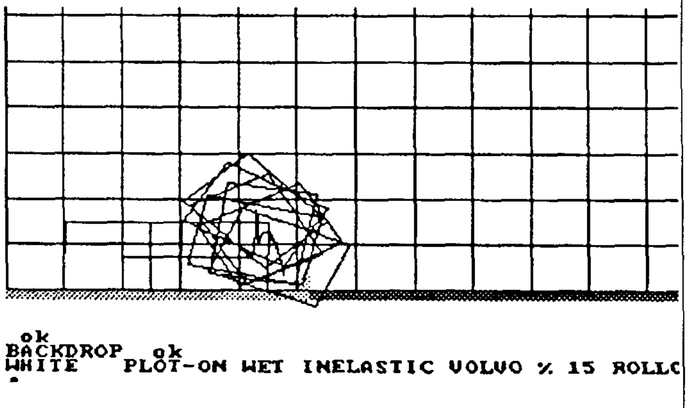
If there is any truth, then, to the old wives' tale, it must pertain to less-extreme situations where the curb is not vertical, but is represented by a change of grade, as

Cross-sectional view of change-of-grade curb:





```
ok
BACKDROP  ok
WHITE   PLOT-ON DRY INELASTIC VOLVO  % 15 ROLLOVER
```



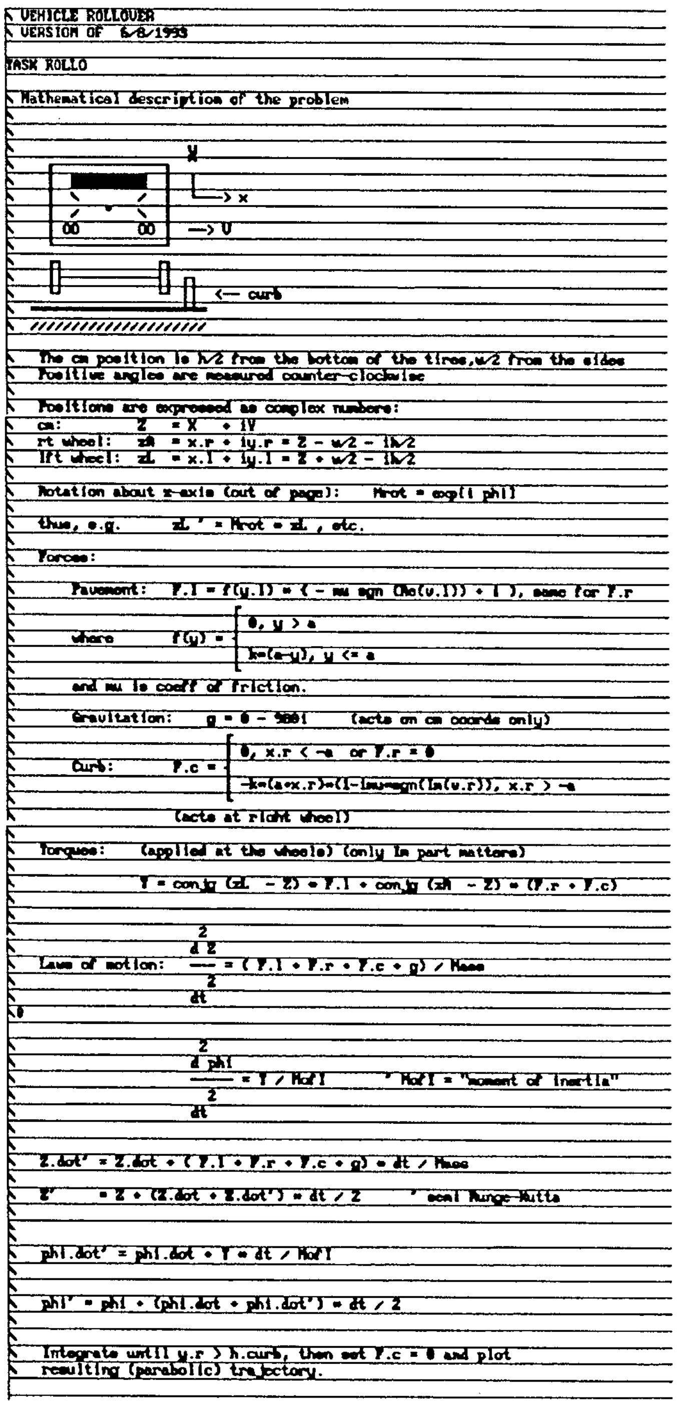
```
ok
BACKDROP  ok
WHITE   PLOT-ON WET INELASTIC VOLVO % 15 ROLLC
```

In addition to the "wet" and "dry" *in*elastic collisions at 15 and 20 mph, we show below an example of a "wet" elastic collision at 20 mph that exhibits the leftward recoil from the barrier mentioned previously.

The Forth Program

Here we exhibit only enough fragments of the Forth program to show what is going on. The program begins with the following logical sections (although it is a single file rather than a set of screens):

- Extensive documentation including character graphics. All the equations and symbols are defined.



```
\ VEHICLE ROLLOVER
\ VERSION OF  6/8/1993

TASK ROLLO

\ Mathematical description of the problem
\
\
\                         y
\
\                          -> x
\
\                          -> v
\
\
\                          <- curb
\
\ ///////////////////////

\  The cm position is h/2 from the bottom of the tires,w/2 from the sides
\  Positive angles are measured counter-clockwise
\
\  Positions are expressed as complex numbers:
\  cm:         Z   = X   + iY
\  rt wheel:   zR  = x.r + iy.r = Z - w/2 - ih/2
\  lft wheel:  zL  = x.l + iy.l = Z + w/2 - ih/2
\
\  Rotation about z-axis (out of page):    Rrot = exp(i phi)
\
\  thus, e.g.      zL ' = Rrot * zL , etc.
\
\  Forces:
\
\      Pavement:  F.l = f(y.l) * ( - mu sgn (Re(v.l)) + i ), same for F.r
\
\                            0, y > a
\      where       f(y) =
\                            k*(a-y), y <= a
\
\      and mu is coeff of friction.
\
\      Gravitation:    g = 0 - 980i     (acts on cm coords only)
\
\                           0, x.r < -a  or F.r = 0
\      Curb:        F.c =
\                           -k*(a+x.r)*(1-imu*sgn(Im(v.r)), x.r > -a
\
\                  (acts at right wheel)
\
\  Torques:   (applied at the wheels) (only Im part matters)
\
\             T = conjg (zL  - Z) * F.l + conjg (zR  - Z) * (F.r + F.c)
\
\
\                    2
\                   d Z
\  Laws of motion:  --- = ( F.l + F.r + F.c + g) / Mass
\                     2
\                   dt
\0
\                    2
\                   d phi
\                   ----- = T / MofI       ' MofI = "moment of inertia"
\                     2
\                   dt
\
\
\  Z.dot' = Z.dot + ( F.l + F.r + F.c + g) * dt / Mass
\
\  Z'     = Z + (Z.dot + Z.dot') * dt / 2      ' ecal Runge-Kutta
\
\
\  phi.dot' = phi.dot + T * dt / MofI
\
\
\  phi' = phi + (phi.dot + phi.dot') * dt / 2
\
\
\  Integrate until y.r > h.curb, then set F.c = 0 and plot
\  resulting (parabolic) trajectory.
```

- Load graphics routines and define display windows.
- Define various data structures. The program needs a great many named SCALARs (generic storage for reals or complex numbers) to hold intermediate results that may be wanted again.
- Define specialized functions FPOS and FSIGN that were not part of the library.

```
\ Documentation (shown above as captured screens)
\ --------------------------------------------------- data structures
\ Define all the data structures
\ ---------------------------------------------- end data structures
FIND F"  0= ?( FLOAD FORTRAN.NEW )
\ make sure FORmula TRANslator is loaded


\ --------------------------------------------------- misc. words
: FPOS   FDUP FABS F+  F2/ ; \ positive part of a #
\ ok 06/07/93
: FSIGN  ( :: x y -- x*sgn[y] )  87type @  FS>87
   87type @  AND  2 AND
   0=  NOT        ABORT" Can't FSIGN complex #"
   F0<  IF  FNEGATE  THEN  ;
\ ok 06/16/1993
\ --------------------------------------------------- end misc. words
```

The next section of the program defines the various forces and torques that can act on the car, as functions of position and velocity. Note that here we start to employ the in-line FORmula TRANslator,[4] i.e., we start to see word definitions of the form

```
: name    F" assignment "
          F" expression "
          etc.    ;
```

```
: tire   ( :: x -- force)   a G@L  FR-  ( :: a-x )
    FPOS  k G@L  F*  ;
: coeff   ( :: v - - 1, v>0 | c)
   F0>  IF  F=1  ELSE  c G@L  THEN  ;
: TIRE.FORCES     \ calculate pavement force on tires
   F" zLdot = Zdot+i*phidot*Rrot*zL "
   F" zLp   = Z + Rrot*zL "
   F" Fl = (i - FSIGN(MU,Re(zLdot)))*tire(Im(zLp)) "
   F" zRdot = Zdot+i*phidot*Rrot*zR "
   F" zRp   = Z + Rrot*zR "
   F" Fr = (i - FSIGN(MU,Re(zRdot)))*tire(Im(zRp)) "
   F" Im( zRp - zC)" FDUP  F0>  FS.DROP
                  \ tire above curb?
   IF    X=0  Fc GIL     ELSE
     F" Fc = (-1-i*FSIGN(MU,Im(zRdot)))*
          tire(-Re(zRp))*coeff(Re(zRdot))"
   THEN  ;
: TORQUE  ( :: -- T)
   F" Im((XCONJG(Rrot*zL)
       *Fl+XCONJG(Rrot*zR)*(Fr+Fc))) "  ;
```

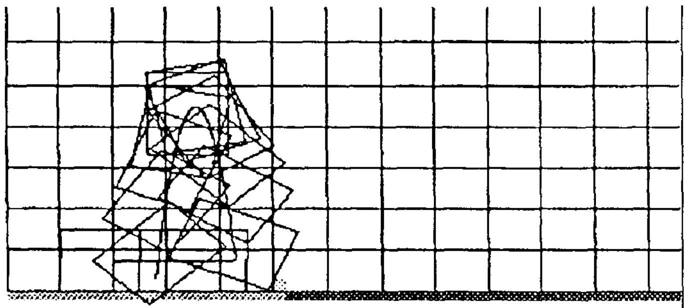
Next there is the section that actually solves the differential equation, and finally, the display section (not shown here). Here are defined words that send either

4. Discussed in detail in my book *Scientific FORTH: a modern language for scientific computing* (Mechum Banks Pubishing, Ivy, VA 1992), ISBN 0-9632775-0-2.

numeric output (time, position, orientation, and velocity) or graphical output (outline of car and plot of its center-of-mass position) to the CRT.

```
: tSTEP  TIRE.FORCES  TORQUE
\ update velocities
   F" dt/MofI"  G*  phidot >FS G+  phidp  FS>
   F" Zdotp = Zdot + (Fl + Fr + Fc + Mg) * dt / M "
\ update positions
   F" Z     = Z + (Zdotp + Zdot) * dt2 "
   F" phi   = phi + (phidp + phidot) * dt2 "
\ update exp[i phi] (rotation matrix)
   F" Rrot = EXP(i * phi) "
\ update old velocities
   F" Zdot  = Zdotp "
   F" phidot = phidp "
\ update time
   F" t = t + dt" ;
```

Elastic collision on a wet surface.



```
 ok
BACKDROP  ok
WHITE PLOT-ON  WET ELASTIC VOLVO % 20 ROLLOVER  ok
```

Why Forth?

One might well ask what Forth has contributed to this study; why would, say, FORTRAN not be as good for solving this problem? The answer, as usual, lies not in the language, *per se*, although it is true that Forth permitted a better factoring than FORTRAN would have, as well as a simpler user interface.

The real advantage of Forth lay in the ease of debugging. As might have been expected, with so many complicated formulae I got several wrong on the first try. The frictional forces were in the wrong direction so they added energy to the system rather than subtracting it. I located and corrected this error rapidly by single-stepping through the problem and displaying results at each successive time increment. To aid this process, I quickly added a calculation of total energy (kinetic and potential) at each step. This let me see how it increased—and why! (Part of the problem came from a subtle error in my FORmula TRANSlator, which this debugging procedure helped me to correct.)

Neither FORTRAN nor any other language I know of (other than Forth, of course!) permits this ease of modification "in flight." To continue the metaphor, even Forth's crashes tend to be soft.

A final remark about execution speed: neither the generic function library nor the ifstack (intelligent floating-point stack) have been optimized in machine code, but were left in high-level Forth (for portability). Nevertheless, the program executes with more-than-adequate celerity on an 80386SX-25 laptop (with numeric co-processor). The numerical integration of the six first-order differential equations (several hundred time steps at intervals $\delta t = 0.01$ sec.) and simultaneous graphical display of the results takes place in real time (i.e., several seconds per case).

Summary

The detailed analysis of an interesting species of automobile collision has given me a new respect for applied mechanics, a discipline often given short shrift in physics or engineering curricula. Programming in Forth greatly reduced the debugging time of this rather complex simulation.

J.V. Noble is Professor of Physics at the University of Virginia. He received his B.S. in Physics at Caltech (1962), and earned his M.A. (1963) and Ph.D. (1966) in Physics at Princeton University. The author of *Scientic Forth*, he also has published in the neighborhood of 100 scientific articles and is currently President of the University of Virginia chapter of the Society of Sigma Xi. He can be reached via e-mail at jvn@fermi.clas.virginia.edu.

dot-quote

Forth has always benefited from its origins in advanced scientific milieus, and this has given it some legitimacy—when we remember to quote and use this fact. But to survive, we also have to look forward and participate in the visions of powerful strategists for the future.

Power being what it is, these go beyond excellence and success. At worst, such visions imply conformance with technically faulty and inadequate standards, crippled systems, etc. Everything that Forth lets you escape from. But in the better cases, understanding which strategies are likely to win provides new, legitimate niches in a rapidly changing world, as in Mitch Bradley's work. In such cases, Forth is able to shine and survive in new roles.

So I ask you to look again at Open Boot and think about how it is more than a success story: it is also well positioned.

Because of Forth's extreme versatility and rapid approach to new hardware (and, I would argue, new software interfaces that should be viewed like new hardware), its strategic potential is enormous.

So, where else can Forth be applied in crucial roles that make use of its unique characteristics, in the mainstream of development?

—David Walker on comp.lang.forth
Adapted with permission

Stretching Forth

LZ77 Data Compression

Wil Baden
Costa Mesa, California

Programmers are lousy lovers. They always try to get the job done faster than before. And when they do, they brag that they have better performance. Programmers are the only men who boast how small theirs is.

Since 1984, there has been amazing progress in data compression. Not so long ago, I got SALIENT SOFTWARE'S AutoDoubler for the Macintosh. My 80-megabyte hard drive had two megs available when I installed the program. Since it was a Tuesday, I went out for lasagna, and when I got back an hour later I had 19 megs available.

My 80-meg hard drive soon held 108 megs worth of data with room for 25 to 50 more megabytes.

Not only that, but many programs loaded faster and read data faster. When a file takes only half as much disk space, the data can be read twice as fast.

How they do it is a trade secret, and Salient has applied for a patent on their technology. There are also many variations possible concerning details.

However, I have a good idea about where to begin looking.

Modern methods of data compression all go back to J. ZIV and A. LEMPEL. In 1977 they published a paper in an engineering journal on a new approach to data compresson.

J. ZIV and A. LEMPEL, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory,* 23:3, 337–343.

In 1978, they published a paper about a related and more elaborate method. In 1984, Unisys employee TERRY WELCH described and had patented a version of the 1978 method suitable for programming. This is called LZW for Lempel, Ziv, and Welch.

LZW is the basis of ARC and PKARC on the PC, *compress* in Unix, and the original StuffIt on the Mac.

Around 1988, after losing a lawsuit, PHIL KATZ (PKARC) came out with a better program, PKZIP. This is derived from the 1977 Ziv-Lempel paper. It turns out that the simpler method has better performance and is smaller. With additional processing, phenomonal results have been obtained.

All popular archivers—arj, lha, zip, zoo, stac, AutoDoubler, current StuffIt—are variations on the LZ77 theme.

The idea of LZ77 is very simple. It is explained in the FAQ (frequently asked question) list for compression technology [see next page]. A copy of this FAQ is available by ftp from rtfm.mit.edu in /pub/usenet/news.answers as compression-faq/part[1-3].

The profane pseudocode given for LZ77 compression can be Forthed as in Figure One.

The bottleneck is the finding the longest match quickly. A naïve brute force method is hardly acceptable. "It's hardly acceptable" is a gentilism for "it sucks". Hashing, or binary search trees, or a combination, is recommended.

A simple implementation of LZSS using binary search trees giving very good, but not best, performance was put into the public domain in 1988 by HARUHIKO OKUMURA. This implementation has inspired the high-performance programs now in use.

Given here is a Standard Forth version of that program. It shows its genealogy by the unusually long Forth definitions. I believe that politically correct factoring would not help understanding and would degrade performance. This program is eight to ten times faster than the brute-force

Figure One. Profane pseudocode.

```
BEGIN
   look-ahead-buffer-used 0= not
WHILE
   get-pointer(position,match)-to-longest-match
   length minimum-match-length > IF
      output-a-(position,match)-pair
      shift-the-window-length-characters-along
   ELSE
      output-first-character-in-lookahead-buffer
      shift-the-window-1-character-along
   THEN
REPEAT
```

* "Standard Forth" is the language defined by ANSI X3.215-1994.

From the LZ77 FAQ.

<The LZ77 family of compressors>

LZ77-based schemes keep track of the last n bytes of data seen, and when a phrase is encountered that has already been seen, they output a pair of values corresponding to the position of the phrase in the previously seen buffer of data, and the length of the phrase. In effect the compressor moves a fixed-size "window" over the data (generally referred to as a "sliding window" [or "ring buffer"], with the position part of the (position, length) pair referring to the position of the phrase within the window. The most commonly used algorithms are derived from the LZSS scheme described by James Storer and Thomas Szymanski in 1982. In this the compressor maintains a window of size N bytes and a "lookahead buffer" the contents of which it tries to find a match for in the window:

```
while( lookAheadBuffer not empty )
    {
    get pointer ( position, match ) to the longest match in the window
        for the lookahead buffer;

    if( length > MINIMUM_MATCH_LENGTH )
        {
        output a ( position, length ) pair;
        shift the window length characters along;
        }
    else
        {
        output the first character in the lookahead buffer;
        shift the window 1 character along;
        }
    }
```

Decompression is simple and fast: Whenever a (position, length) pair is encountered, go to that (position) in the window and copy (length) bytes to the output.

Sliding-window-based schemes can be simplified by numbering the input text characters mod N, in effect creating a circular buffer. The sliding window approach automatically creates the LRU effect which must be done explicitly in LZ78 schemes. Variants of this method apply additional compression to the output of the LZSS compressor, which include a simple variable-length code (LZB), dynamic Huffman coding (LZH), and Shannon-Fano coding (ZIP 1.x), all of which result in a certain degree of improvement over the basic scheme, especially when the data are rather random and the LZSS compressor has little effect.

Patent history.

- Waterworth patented (4,701,745) the algorithm now known as LZRW1, because Ross Williams reinvented it later and posted it on comp.compression on April 22, 1991. The *same* algorithm has later been patented by Gibson & Graybill. The patent office failed to recognize that the same algorithm was patented twice, even though the wording used in the two patents is very similar.

  The Waterworth patent is now owned by Stac Inc., which won a lawsuit against Microsoft, concerning the compression feature of MSDOS 6.0. Damages awarded were $120 million.

- Fiala and Greene obtained in 1990 a patent (4,906,991) on all implementations of LZ77 using a tree data structure.

- Notenboom (from Microsoft) 4,955,066 uses three levels of compression, starting with run-length encoding.

- The Gibson & Graybill patent 5,049,881 covers the LZRW1 algorithm previously patented by Waterworth and reinvented by Ross Williams. Claims 4 and 12 are very general and could be interpreted as applying to any LZ algorithm using hashing (including all variants of LZ78):

- Phil Katz, author of pkzip, also has a patent on LZ77 (5,051,745) but the claims only apply to sorted hash tables, and when the hash table is substantially smaller than the window size.

- IBM patented (5,001,478) the idea of combining a history buffer (the LZ77 technique) and a lexicon (as in LZ78).

- Stac Inc. patented (5,016,009 and 5,126,739) yet another variation of LZ77 with hashing. The '009 patent was used in the lawsuit against Microsoft (see above). Stac also has patents on LZ77 with parallel lookup in hardware (4,841,092 and 5,003,307).

- Chambers 5,155,484 is yet another variation of LZ77 with hashing. The hash function is just the juxtaposition of two input bytes. This is the 'invention' being patented. The hash table is named 'direct lookup table.' [Chambers is the author of AutoDoubler and DiskDoubler.]

implementation I gave at the 1992 FORML Conference. It can serve as material for studying data compression in Forth, as the original program did in C and Pascal.

As an example, here is the beginning of *Green Eggs and Ham*, copyright 1960, Dr. Seuss.

```
That Sam-I-am!
That Sam-I-am!
I do not like that Sam-I-am!

Do you like green eggs and ham?

I do not like them, Sam-I-am.
I do not like green eggs and ham.
```

Compressed with LZSS this becomes:

```
|That Sam|-I-am!
[]|I do not| like t[]|
Do you[]|green eg|gs and h|am?
[]em,|[].[][].
```

"|" represents a format byte. "[]" represents a two-byte position and length.

The program uses words from the Core and Core Extension wordsets. It also uses READ-FILE and WRITE-FILE from the File Access word set. It presumes that R/O, R/W, W/O, BIN, OPEN-FILE, CREATE-FILE, and TO will be used appropriately for file assignment.

The program also uses not, which can be equivalent to either 0= or INVERT.

Standard Forth file access for character-by-character input or output is hardly acceptable. read-char used here can be painfully defined with READ-FILE. (See Figure Two.)

Standard words are written without lower-case letters. Non-standard words contain one or more lower-case letters or are single-letter, upper-case words other than I or J. The spelling of a word is consistent and no words are distinguished by a difference of case. It is immaterial whether letter-case in your system is significant or insignificant.

Definitions of LZSS-Data-Compression and reload have been commented out. They were used during development.

Figure Two. File-access definitions.

```
: checked   ABORT" File Access Error. " ;     ( ior -- )

CREATE single-char-i/o-buffer    0 C,    ALIGN

: read-char                           ( file -- char )
    single-char-i/o-buffer 1 ROT READ-FILE checked IF
        single-char-i/o-buffer C@
    ELSE
        -1
    THEN
;
```

A better definition would be to buffer input of many characters at a time.

Note: The definition in ThisForth is a macro.

```
: read-char
    please "stream get-char unstream "
; IMMEDIATE
```

In ThisForth, macro-defining definitions for array and carray improve performance 25 percent.

```
: array
    CREATE CELLS ALLOT IMMEDIATE
    DOES> (.) please "CELLS ~ + "
;

: carray
    CREATE CHARS ALLOT IMMEDIATE
    DOES> (.) please "CHARS ~ + "
;
```

P.S. I no longer go out for lasagna on Tuesday, but if you come to my house, I know where to get some great Italian or Mexican food.

Wil Baden is a professional programmer with an interest in Forth. He can be contacted at his wilbaden@netcom.com e-mail address.

Code begins on next page...

Baden's LZSS Forth code.

```
(     LZSS -- A Data Compression Program )
(     89-04-06 Standard C by Haruhiko Okumura )
(     94-12-09 Standard Forth by Wil Baden )

(     Use, distribute, and modify this program freely. )

\ MARKER      LZSS-Data-Compression

\ : reload    LZSS-Data-Compression    S" lzss.fo" INCLUDED ;

4096  CONSTANT   N      ( Size of Ring Buffer )
18    CONSTANT   F      ( Upper Limit for match-length )
2     CONSTANT   Threshold ( Encode string into position & length
                        ( if match-length is greater. )
N     CONSTANT   Nil    ( Index for Binary Search Tree Root )

VARIABLE    textsize     ( Text Size Counter )
VARIABLE    codesize     ( Code Size Counter )
\ VARIABLE  printcount   ( Counter for Reporting Progress )

( These are set by InsertNode procedure. )

VARIABLE    match-position
VARIABLE    match-length

: array         CREATE  CELLS ALLOT     DOES>   SWAP CELLS + ;

: carray        CREATE  CHARS ALLOT     DOES>   SWAP CHARS + ;

N F + 1 -    carray text-buf    ( Ring buffer of size N, with extra
                   ( F-1 bytes to facilitate string comparison. )

( Left & Right Children and Parents -- Binary Search Trees )

N 1 +        array lson
N 257 +      array rson
N 1 +        array dad

( Input & Output Files )

0 VALUE     infile      0 VALUE     outfile

( For i = 0 to N - 1, rson[i] and lson[i] will be the right and
( left children of node i.  These nodes need not be initialized.
( Also, dad[i] is the parent of node i.  These are initialized to
( Nil = N, which stands for `not used.'
( For i = 0 to 255, rson[N + i + 1] is the root of the tree
( for strings that begin with character i.  These are initialized
( to Nil.  Note there are 256 trees. )

( Initialize trees. )

: InitTree                                 ( -- )
      N 257 +  N 1 +  DO    Nil   I rson !    LOOP
      N  0  DO    Nil  I dad !    LOOP
;

( Insert string of length F, text-buf[r..r+F-1], into one of the
( trees of text-buf[r]'th tree and return the longest-match position
( and length via the global variables match-position and
( match-length.  If match-length = F, then remove the old node in
( favor of the new one, because the old one will be deleted sooner.
( Note r plays double role, as tree node and position in buffer. )

: InsertNode                               ( r -- )

      Nil OVER lson !    Nil OVER rson !    0 match-length !
      DUP text-buf C@  N + 1 +                 ( r p)

      1                                        ( r p cmp)
      BEGIN                                    ( r p cmp)
```

```
            0< not IF                           ( r p)

                  DUP rson @ Nil = not IF
                        rson @
                  ELSE

                        2DUP rson !
                        SWAP dad !              ( )
                        EXIT

                  THEN
            ELSE                                ( r p)

                  DUP lson @ Nil = not IF
                        lson @
                  ELSE

                        2DUP lson !
                        SWAP dad !              ( )
                        EXIT

                  THEN
            THEN                                ( r p)

            0 F DUP 1 DO                        ( r p 0 F)

                  3 PICK I + text-buf C@        ( r p 0 F c)
                  3 PICK I + text-buf C@ -      ( r p 0 F diff)
                  ?DUP IF
                        NIP NIP I
                        LEAVE
                  THEN                          ( r p 0 F)

            LOOP                                ( r p cmp i)

            DUP match-length @ > IF

                  2 PICK match-position !
                  DUP match-length !
                  F < not

            ELSE
                  DROP FALSE
            THEN                                ( r p cmp flag)
      UNTIL                                     ( r p cmp)
      DROP                                      ( r p)

      2DUP dad @ SWAP dad !
      2DUP lson @ SWAP lson !
      2DUP rson @ SWAP rson !

      2DUP lson @ dad !
      2DUP rson @ dad !

      DUP dad @ rson @ OVER = IF
            TUCK dad @ rson !
      ELSE
            TUCK dad @ lson !
      THEN                                      ( p)

      dad Nil SWAP !    ( Remove p )            ( )
;

( Delete node p from tree. )

: DeleteNode                              ( p -- )

      DUP dad @ Nil = IF    DROP EXIT    THEN   ( Not in tree. )

      ( CASE )                                  ( p)
            DUP rson @ Nil =
      IF
```

```
            DUP lson @
      ELSE
            DUP lson @ Nil =
      IF
            DUP rson @
      ELSE

            DUP lson @                         ( p q)

            DUP rson @ Nil = not IF

                  BEGIN
                        rson @
                        DUP rson @ Nil =
                  UNTIL

                  DUP lson @ OVER dad @ rson !
                  DUP dad @ OVER lson @ dad !

                  OVER lson @ OVER lson !
                  OVER lson @ dad OVER SWAP !
            THEN

            OVER rson @ OVER rson !
            OVER rson @ dad OVER SWAP !

      ( 0 ENDCASE ) THEN THEN                       ( p q)

      OVER dad @ OVER dad !

      OVER DUP dad @ rson @ = IF
            OVER dad @ rson !
      ELSE
            OVER dad @ lson !
      THEN                                     ( p)

      dad Nil SWAP !                           ( )
;

      17 carray   code-buf

      VARIABLE    len
      VARIABLE    last-match-length
      VARIABLE    code-buf-ptr

      VARIABLE    mask

: Encode                                  ( -- )

      InitTree    ( Initialize trees. )

      ( code-buf[1..16] holds eight units of code, and code-buf[0]
      ( works as eight flags, "1" representing that the unit is an
      ( unencoded letter in 1 byte, "0" a position-and-length pair
      ( in 2 bytes.  Thus, eight units require at most 16 bytes
      ( of code. )

      0 0 code-buf C!
      1 mask C!   1 code-buf-ptr !
      0    N F -                               ( s r)

      ( Clear the buffer with a character that will appear often. )

      0 text-buf  N F -  BL  FILL

      ( Read F bytes into the last F bytes of the buffer. )

      DUP text-buf F infile READ-FILE checked   ( s r count)
      DUP len !    DUP textsize !
      0= IF    2DROP EXIT    THEN               ( s r)
```

```
( Insert the F strings, each of which begins with one or more
( `space' characters.  Note the order in which these strings
( are inserted.  This way, degenerate trees will be less
( likely to occur. )

F 1 + 1 DO                              ( s r)
      DUP I - InsertNode
LOOP

( Finally, insert the whole string just read.  The global
( variables match-length and match-position are set. )

DUP InsertNode

BEGIN                                   ( s r)

      ( match-length may be spuriously long at end of text. )
      match-length @ len @ > IF    len @ match-length !   THEN

      match-length @ Threshold > not IF

            ( Not long enough match.  Send one byte. )
            1 match-length !
            ( `send one byte' flag )
            mask C@ 0 code-buf C@ OR 0 code-buf C!
            ( Send uncoded. )
            DUP text-buf C@ code-buf-ptr @ code-buf C!
            1 code-buf-ptr +!

      ELSE
            ( Send position and length pair. )
            ( Note match-length > Threshold. )

            match-position @  code-buf-ptr @ code-buf C!
            1 code-buf-ptr +!

            match-position @  8 RSHIFT  4 LSHIFT ( . . j)
                  match-length @  Threshold - 1 -  OR
                  code-buf-ptr @  code-buf C!  ( . .)
            1 code-buf-ptr +!

      THEN

      ( Shift mask left one bit. )        ( . .)

      mask C@  2*  mask C!    mask C@ 0= IF

            ( Send at most 8 units of code together. )

            0 code-buf  code-buf-ptr @    ( . . a k)
                  outfile WRITE-FILE checked ( . .)
            code-buf-ptr @  codesize  +!
            0 0 code-buf C!    1 code-buf-ptr !    1 mask C!

      THEN                                ( s r)

      match-length @ last-match-length !

      last-match-length @ DUP 0 DO        ( s r n)

            infile read-char              ( s r n c)
            DUP 0< IF   2DROP I LEAVE   THEN

            ( Delete old strings and read new bytes. )

            3 PICK DeleteNode
            DUP 3 1 + PICK text-buf C!

            ( If the position is near end of buffer, extend
            ( the buffer to make string comparison easier. )
```

```
                    3 PICK F 1 - < IF            ( s r n c)
                          DUP 3 1 + PICK N + text-buf C!
                    THEN
                    DROP                         ( s r n)

                    ( Since this is a ring buffer, increment the
                    ( position modulo N. )

                    >R >R                        ( s)
                          1 +    N 1 - AND
                    R>                           ( s r)
                          1 +    N 1 - AND
                    R>                           ( s r n)

                    ( Register the string in text-buf[r..r+F-1]. )

                    OVER InsertNode

              LOOP                               ( s r i)
              DUP textsize +!

              \ textsize @  printcount @ > IF

              \     ( Report progress each time the textsize exceeds
              \     ( multiples of 1024. )
              \     textsize @ 12 .R
              \     1024 printcount +!

              \ THEN

              ( After the end of text, no need to read, but
              ( buffer might not be empty. )

              last-match-length @ SWAP ?DO       ( s r)

                    OVER DeleteNode

                    >R  1 +  N 1 - AND  R>
                    1 +  N 1 - AND

                    -1 len +!    len @ IF
                          DUP InsertNode
                    THEN
              LOOP

              len @ 0> not
        UNTIL                                    2DROP

        ( Send remaining code. )

        code-buf-ptr @ 1 > IF
              0 code-buf  code-buf-ptr @  outfile WRITE-FILE checked
              code-buf-ptr @ codesize +!
        THEN
;

: Statistics                            ( -- )
        ." In : "   textsize ?   CR
        ." Out: "   codesize ?   CR
        textsize @ IF
              ." Saved: " textsize @ codesize @ - 100 textsize @ */
                    2 .R ." %" CR
        THEN
        infile closed    outfile closed
;
```

```
( Just the reverse of Encode. )

: Decode                                   ( -- )

      0 text-buf  N F -  BL FILL

      0  N F -                                   ( flags r)
      BEGIN
            >R                                   ( flags)
                  1 RSHIFT DUP 256 AND 0= IF DROP      ( )
                        infile read-char         ( c)
                        DUP 0< IF                R> 2DROP
                              EXIT               ( c)
                        THEN
                        [ HEX ] 0FF00 [ DECIMAL ] OR ( flags)
                        ( Uses higher byte to count eight. )
                  THEN
            R>                                   ( flags r)

            OVER 1 AND IF

                  infile read-char               ( . . c)
                  DUP 0< IF                      DROP 2DROP
                        EXIT                     ( . r c)
                  THEN

                  OVER text-buf C!               ( . r)
                  DUP text-buf 1 outfile WRITE-FILE checked

                  1 +    N 1 - AND

            ELSE

                  infile read-char               ( . . i)
                  DUP 0< IF                      DROP 2DROP
                        EXIT                     ( . r i)
                  THEN

                  infile read-char               ( . . i j)
                  DUP 0< IF                      2DROP 2DROP
                        EXIT                     ( . . i j)
                  THEN

                  DUP >R    4 RSHIFT    8 LSHIFT OR    R>
                  15 AND    Threshold +    1 +

                  0 ?DO                          ( . r i)

                        DUP I +  N 1 - AND  text-buf ( . r i a)
                        DUP 1 outfile WRITE-FILE checked
                        C@  2 PICK text-buf C!  ( . r i)
                        >R  1 +  N 1 - AND  R>

                  LOOP                           ( . r i)
                  DROP                           ( flags r)

            THEN
      AGAIN
;
```

PowerMacForth Optimizer

Xan Gregg
Durham, North Carolina

Several past *Forth Dimensions* articles have extolled the virtues of an optimizing direct-code Forth compiler, so I'll try not to repeat those arguments. Instead, I will present the implementation of such a compiler on the PowerPC. This particular implementation is Creative Solutions, Inc.'s Power MacForth, an ANS version of their venerable MacForth product for the PowerMacintosh.

Forth on the PowerPC

The PowerPC 601 is a RISC microprocessor which includes 32 32-bit, general-purpose registers, a load-store architecture, and fixed-length instructions. The Mac OS takes up only a few registers, leaving Forth plenty for TOS, a handful of pointers, eight locals, and a few spares. The load-store architecture does make some stack operations more painful than on CISC machines, but at least the PowerPC allows a pre-increment addressing mode, which makes possible a one-instruction push (by using a negative increment amount). The fixed-length instruction are, of course, a boon to optimizers and decompilers.

...it keeps the compiler relatively simple and puts the intelligence in the data structures.

Another useful feature of the PowerPC is the "branch-folding" which occurs when the pre-fetch unit sees a branch instruction in the instruction queue and can resolve the branch before the instructions preceding it are even executed. Subroutine calls are just unconditional branches that stash the return address in the "link register" before branching, and being unconditional, they are predictable as soon as they enter the instruction queue. The advantage for Forth is that subroutines are, overall, very cheap instructions—in fact, they are free if preceded by two or three non-branch instructions (like an in-lined stack manipulation word).

PowerMacForth

PowerMacForth, also known as "MacForth 5.0 for the Power Macintosh," is a 32-bit ANS Forth. It includes all or most words in the following wordsets: Core, Core Ext, Exception, Exception Ext, Search, Search Ext, and String. Floating, Memory, and File are included in source code form for optional use. Local variables are implemented with `LOCALS|`, `TO`, and `+TO`, and are stored in up to eight machine registers.

PowerMacForth has separate code and data areas. This helps tools that examine code, since the code area contains only 32-bit PowerPC instructions and no data. It is also possible to make the code area a read-only segment in the future for turnkeys. Execution tokens, usually referred to as just "tokens," are 32-bit offsets within the code area. Similarly, data locations are sometimes referenced with "data offsets," which are offsets from the "data base pointer," which is an address within the data area. Vocabularies ("Wordlists" in ANS-speak) are also stored in separate memory areas. They are relocatable and grow automatically as needed, and they are discarded for turnkeys. MacForth has used hashed vocabularies since version 4.0 for very fast searches.

Being a direct-code compiler, every Forth word contains callable machine code in its code space. Colon definitions begin with `NEST` code to push the return address onto the return stack, and end with `UNNEST` code to pop it off before returning. A `VARIABLE` word contains code to push the address of its data onto the data stack. A `CONSTANT` word contains code to push its value onto the data stack. As a consequence, `CONSTANT`s have no data and cannot be easily hot-patched.

A View From the Top

The PowerMacForth optimizing compiler was designed to have a simple structure with as much of the "smarts" as possible being contained in data structures that could be jettisoned when building a turnkey application. It was also built with flexibilty in mind, so that the user could add more optimizations or less. This was accomplished by making many of the compiler words `DEFER` words, which are easily patchable with `IS`.

INTERPRET is a DEFER word itself. Here is its default action in PowerMacForth:

```
: DFLTINTERPRET
  ( -- | interpret or compile input text )
   BEGIN BLWORD C@
   WHILE GET-ORDER POCKET #FIND
         ?TRACE ?DUP
         IF    INTERPRET-TOKEN
         ELSE    INTERPRET-UNKNOWN-WORD
         THEN
         INTERPRET-CHECKER
   REPEAT ;
```

INTERPRET-TOKEN is also a DEFER word. Its default behavior is to either execute the word or call CALLTOKEN, on it. (The comma is part of the name and indicates that the word will "comma" some code into the code area.) CALLTOKEN, is (what else) a DEFER word as well. Its default action will either inline the word or compile a call to it and, in either case, possibly invoke a pattern reduction on the new code.

DO-UNKNOWN-WORD is an interesting feature that has nothing to do with the optimizer. It walks down an extensible "chain" of words that try to interpret the unknown text until it gets to a handler that can. If no handler recognizes the text, an exception is generated. The kernel includes a handler that interprets numbers. The floating-point package adds another handler to the chain to interpret floating-point numbers. It would be possible to add handlers for such purposes as C-style hex numbers (0xFF) or character constants ('A').

The Optimizations

The PowerMacForth compiler can do four types of optimization. If a word to be called is small enough, it will be inlined instead of called. The threshold is contained in a variable and is initially 12 bytes (three instructions). Words can be flagged as non-inlineable, however.

The compiler keeps a history of recently compiled words. After each addition to the history, the history is compared against a table of reduction patterns. If a match is found, the pattern is substituted for new, more efficient code.

If a colon definition doesn't call any other words (because all of its words are inlined), the colon definition is made into a code definition, meaning it is stripped of its NEST and UNNEST code. Such a word is a "leaf" word, and the optimization is call "leafing." It is important not only for speed, but also because the NEST and UNNEST code adds five instructions (20 bytes) to a definition.

The final optimization is called "chaining." It occurs when a colon definition ends with a call to another colon definition. In that case, the caller's UNNEST and the callee's NEST can be skipped. The caller's UNNEST may still need to be compiled if there was a pending branch to the end of the word, but, in any case, the call instruction can be compiled so that it jumps into the callee just past the NEST code and does not return to the caller. Chaining allows infinite tail-recursion without overflowing the return stack.

Inlining and reducing are invoked during CALLTOKEN,, while leafing and chaining are invoked during ;. Each of these may be turned off separately via a bit in the OPTIMIZER variable, which is initially all ones.

CodeInfo Table

The CodeInfo table is a data structure used by the compiler; it is not needed at run-time, so it can be thrown away for turnkeys. The table contains a few items about every word in the code area. The entries are in a sequential list, sorted by token, which makes it possible for a token's entry to be located quickly with a binary search. The fields of each entry are as follows:

field	size	description
token	4	word's execution token
value	4	data offset for data words, value for constants
type	1	one of several common type values, or zero for "unknown"
flags	1	inlineable flag and possibly others
action	2	CodeInfo index of action word for DOES words
offset	4	current data offset when this word was created

The CodeInfo table is used in several ways by the optimizer. Since entries are consecutive words, a word's size may be determined by subtracting the next token from the current token. The size is useful to know when determining whether to inline or not, as is the inlineable bit in the flags field. The value field allows the optimizer to know the value of a constant or the offset of a variable without having to look at the machine code for the word.

The type field is used during chaining to know that the callee is a colon definition. And it is used when adding a word to the compiler history, since constant words and literals will look the same in the history, as will variables and other words that return a data address. The action and offset fields are primarily intended to be used by decompilers and other tools that examine or move words (or remove them, like FORGET).

Like the code size, the size of a word's data can be determined by subtracting its data offset from the next word's data offset. A word's data offset is determined by subtracting the data base pointer from HERE at the time the word is created. Note that even colon words can have data if they use string literals.

Compiler History

Whenever a word is compiled, the compiler adds information about the compiled word into its "history." The history buffer is limited to five entries, and each entry has the following four fields:

Figure One. Example word counts lines in a text block.

```
: COUNT-LINES   ( start\count -- #lines )
  OVER + LOCALS| START END |   \ start also serves as running ptr
  0                            \ the initial count
  BEGIN
     START END <               \ check for end of text
  WHILE
     START C@ 13 = IF 1+ THEN  \ CR => bump line count
     1 +TO START               \ go to next character
  REPEAT ;
```

field	size	description
token	4	negative for special cases (literals, etc.)
addr	4	code address where compiled
value1	4	literal value, variable offset, etc.
value2	4	literal value, variable offset, etc.

Odd negative tokens are accompanied by something in the value1 field, and even negative tokens have data in both value fields. The compiler will convert token for items like constants, variables, and locals into the appropriate negative token and value. (It does that by using the type and value fields from the CodeInfo table). The second value field is useful when optimizing doubles, floats (also eight bytes), or just combinations of two single-value entries.

The word ADD-INFOENTRY-TO-HISTORY does the work of adding a word properly to the history. It is called by the CALLTOKEN, action before each word that is compiled into a colon definition, whether inlined or called. The "InfoEntry" in the name refers to the fact that CALLTOKEN, has already looked up the word in the CodeInfo table (to see if it is inlineable), and is passing the entry number within the table.

It has to be fast, because a search is made for almost every word compiled...

Reduction Patterns

A reduction pattern consists of a list of tokens (the "pattern") plus a replacement token. The replacement token is either itself compiled as a replacement for the pattern, or it is executed to compile code to replace the pattern. The former occurs when the tokens in the pattern are all positive (no value fields used in the history entries). An example of such a pattern is SWAP and DROP with the replacement token of NIP.

When a token in the pattern is negative, the replacement token is executed instead of compiled. An example of that would be the pattern <LIT>, <LIT>, and + with the replacement token of LITERAL+, (which is definied as simply "+ LITERAL, "). When the replacement token executes, the values associated with all of the pattern tokens are on the stack. The reducer knows how many items to push, based on whether the token is non-negative (no values), odd negative (one value), or even negative (two values). So, in this example (because <LIT> is a constant with an odd negative value), the two literals would be on the stack when LITERAL+, is called.

The work is done by the word ?REDUCE-HISTORY which is called by CALLTOKEN, after each word it compiles. ?REDUCE-HISTORY looks up in the list of reduction patterns all patterns in the history that end with the newest entry, from longest to shortest. So, if the history was "A B C D", ?REDUCE-HISTORY would first look up "A B C D", then "B C D", and finally "C D" until it found a reduction pattern. If a pattern is found, the code pointer is moved back to where the pattern was compiled and the replacement is either compiled or executed, based on whether any of the history entries were negative pseudo-tokens.

The difficulty is how to store a list of variable-length reduction patterns so they can be searched quickly, given a list of tokens from the history. It has to be fast, because a search is made for almost every word compiled (whenever there are at least two items in the history). The best approach, initially, seemed to be a list of sorted, fixed-length patterns. This involved a little wasted memory, since most patterns are only two or three tokens long and a little extra code for the insertion and searching.

However, CSI's Ward McFarland came up with the great suggestion of using a vocabulary to store the patterns. If the list of tokens is treated as a string and the replacement token is treated as the value, the reduction patterns can be considered a list of strings with corresponding values. And that happens to be the general structure for the hashed, MacForth vocabularies, where normally the string is a Forth name and the value is its token.

So how does a list of tokens get treated as a string? The tokens are just stored one after another in memory (actually on the return stack), and the list is preceded by a length byte. Since each token is four bytes, the length byte is 4 * numTokens, where numTokens is the number of tokens in the list. Fortunately, the vocabulary mechanism is general enough that it allows strings with characters that are any byte value, including zero.

With reduction patterns made to look like vocabulary entries, it was easy to use the vocabulary words for insertion, look-up, and removal of patterns. And the criterion for speed was well-satisfied, since the vocabularies are organized with a hashing function.

Table One. History of compiler actions.

	Word	Action	Compiler History
1	:	compile code for NEST	empty
2	OVER	inline code for OVER	OVER
3	+	inline code for +	OVER +
4		reduce "OVER +"	OVER+
5	LOCALS|	compile call to PUSH2LOCALS	OVER+ PUSH2LOCALS
6	0	compile code for literal	OVER+ PUSH2LOCALS <LIT>
7	BEGIN	no code, clear history	empty
8	START	inline code for LOCAL1	<LOCAL>
9	END	inline code for LOCAL2	<LOCAL> <LOCAL>
10	<	inline code for <	<LOCAL> <LOCAL> <
11		reduce "<LOCAL> <LOCAL> <"	<LOCALLOCAL-> 0<
12	WHILE	compile code for 0BRANCH	<LOCALLOCAL-> 0< <0BRANCH>
13		reduce "<LOCALLOCAL-> 0< <0BRANCH>"	empty
14	START	inline code for LOCAL1	<LOCAL>
15	C@	inline code for C@	<LOCAL> C@
16		reduce "<LOCAL> C@"	empty
17	13	compile code for literal	<LIT>
18	=	compile call for =	<LIT> =
19		reduce "<LIT> ="	<LIT+> 0=
20	IF	compile code for 0BRANCH	<LIT+> 0= <0BRANCH>
21		reduce "<LIT+> 0= <0BRANCH>"	empty
22	1+	compile code for literal	<LIT>
23		inline code for +	<LIT> +
24		reduce "<LIT> +"	<LIT+>
25	THEN	resolve branch, clear history	empty
26	1	compile code for literal	<LIT>
27	+TO	compile +>LOCAL1	<LIT> <+>LOCAL>
28		reduce "<LIT> <+>LOCAL>"	empty
29	REPEAT	resolve BEGIN & WHILE	empty
27	;	check for Leafing or Chaining optimizations	
28		compile code for UNNEST and return instruction	

Example

Now that the data structures have been presented, let's see how it all fits together with an example. Figure One shows a word that counts the number of lines in a block of text.

Table One shows the compiler history and the compiler's action for each word interpreted. The words in angle brackets, such as <LIT>, are constants for negative tokens. When such entries occur in the compiler history, they are accompanied by one or two values which are not shown in the table. For instance, in step six, the history entry for <LIT> also includes the value 0, and, in step eight, the history entry for <LOCAL> also contains the value 1 since START is local #1.

START and END are added to the history as locals because their entries in the CodeInfo table indicates that their types are both Local.

Line four shows a simple reduction where the code for the pattern "OVER +" is replaced by the code for the single word OVER+. That reduces five instructions down to three. This is the classic kind of ideal reduction, because it merges a producer, OVER, with a consumer, +, into one piece of code that does not alter the stack depth.

The next reduction occurs on line eleven and is actually more useful for its regrouping as much as for its code optimization. Essentially, the sequence "START END <" is converted to "START END - 0<". The code is better and avoids special handling of the comparison, whose result will probably just be consumed by a 0BRANCH, as it is here. Note that the pseudo-token <LOCALLOCAL-> is an even negative number, as it carries two values with it, one for each of the locals.

When the 0BRANCH does follow, it reduces the sequence to a simple register-to-register compare and a branch. Later, in line 21, a similar reduction is made for a literal comparison.

Note how 1+ is handled, starting on line 21. 1+ is actually an immediate word that is equivalent to "1 +" (two words). That way, the optimizer sees it as a "<LIT> +" sequence instead of some unary arithmetic function to be special-cased. However, it does no good here, because there is nothing before or after the 1+ that it can be combined with for optimization.

Without further ado, the annotated code produced by the compiler is shown in Figure Two.

It's actually one-third smaller than the unoptimized

Figure Two. Optimized code generated by the compiler.

```
      mfspr   r11/X, LR                     code for NEST...
      stwu    r11/X, $-4(r15/RSP)            ... push LR to return stack
      lwz     r11/X, (r14/DSP)              code for OVER+
      add     r13/TOS, r13/TOS, r11/X        ...
      bl      PUSH2LOCALS                   call PUSH2LOCALS
      stwu    r13/TOS, $-4(r14/DSP)         make room for new TOS
      addi    r13/TOS, 0, 0                 literal to TOS
L1:   cmp     r30/Local2, r31/Local1        START < END ?
      bge     L3                            exit loop if false
      stwu    r13/TOS, $-4(r14/DSP)         make room for new TOS
      lbz     r13/TOS, (r30/Local2)         START C@ -> TOS
      cmpi    r13/TOS, 13                   = 13 ?
      lwz     r13/TOS, (r14/DSP)            pop new TOS...
      addi    r14/DSP, r14/DSP, 4            ... gives time to resolve branch
      bne     L2                            branch for IF
      addi    r13/TOS, r13/TOS, 1           1+
L2:   addi    r30/Local2, r30/Local2, 1     1 +TO START
      b       L1                            REPEAT
L3:   lwz     r11/X, (r15/RSP)              UNNEST...
      mtspr   LR, r11/X                      ...
      addi    r15/RSP, r15/RSP, 4            ...
      blr
```

code, as well as 4.8 times faster. COUNT-LINES was derived from code in MacForth's integrated editor. In the token-threaded, 68K version of MacForth, this word and similar ones had to be written in assembly for adequate performance on large text files; but with the faster processor and optimizing compiler of PowerMacForth, it was fine to use high-level Forth.

This example only scratches the surface of the number of optimizations possible, of course. PowerMacForth ships with 169 reduction patterns. 42 of those deal with 0BRANCH, 57 deal with locals, and most of the others involve fetching and storing. More optimizations are added by the floating-point package.

Conclusion

The PowerMacForth optimizing compiler keeps with Forth tradition by keeping the compiler relatively simple and putting the intelligence in the data structures. Of practical importance is that the compiler data structures are in separate memory blocks which are not retained in a turnkey. The compiler is extensible by changing the action of compiler defer words, adding unknown word handlers, or adding reduction patterns.

Readers can contact Xan Gregg at his xgregg@aol.com e-mail address. He is a freelance Macintosh programmer living in Durham, North Carolina, who mainly writes printer drivers and medical-imaging software. Xan has been programming with MacForth since 1984 and, in his free time, he plays ultimate Frisbee.

ADVERTISERS INDEX

Forth Interest Group	centerfold
Laboratory Microsystems, Inc.	19
Offete Enterprises	19
Miller Microcomputer Services	17
Ruffer Graphic Solutions	21
Silicon Composers	2

MuP21: a High-Performance MISC Processor

Chen-hanson Ting, Charles H. Moore
San Mateo, California

MISC vs. RISC vs. CISC

The controversy between RISC (Reduced Instruction Set Computer) and CISC (Complicated Instruction Set Computer) had pretty much settled, and RISC had won. Most newer and more powerful processors developed recently are all RISC processors, like SPARC, MIPS, Alpha from DEC, PA from H-P, and PowerPC from IBM. However, CISC processors persist due to momentum, like the Intel x86 family, and in the microcontroller area where raw speed is not an important factor.

The basic principles behind the original RISC processors are valid, such as:

a. A simple instruction set is faster.
b. Complicated memory-accessing instructions are not necessary.
c. A large register file facilitates software.
d. Complicated functions are best handled in compiler.
e. A simpler processor is easier to design and to build.

However, RISC is a good idea falling into the wrong hands. The emphasis on simplicity is all but forgotten. The RISC processors we see now are more complicated than many of the CISC processors. The relentless push towards higher speed left a bloody trail. Some of the problems in the RISC architecture are quite evident:

a. RISC processors are inherently slow, because each instruction still needs many machine cycles to execute. An instruction pipeline is used to accelerate the execution; however, the pipeline must be flushed and refilled when a branch instruction is encountered.
b. Increasing speed in the RISC processor creates a large disparity between the processor and the slower memory. To increase the memory-accessing speed, it is necessary to use cache memory to buffer instruction and data streams. The cache memory brings in a whole set of problems, which complicates the system design and renders the system more expensive.
c. RISC processors are very inefficient in handling subroutine calls and returns. An efficient subroutine mechanism is critical to the performance of a processor in supporting high-level languages. Many RISC processors use a large register file, which is windowed to facilitate subroutine call and return. However, the register window must be big enough to handle a large set of input, output, and local parameters. The large register window wastes the most precious resource in the RISC processor. A large register file also slows down the system during a context switch, which must save the register file and later restore it.

Our opinion is that, in RISC, reducing the size of the instruction set is effective in reducing the complexity of the processor and improving its performance. However, the principle of simplicity was not enforced well enough to realize the full benefits of this principle. In the MISC architecture, we like to explore the power of simplicity to its limit, to see how far we can push the CMOS technology in reducing the costs of building computer systems and increasing their performance. We like to have answers to the following questions:

a. What is the minimum set of instructions in a microprocessor to make it useful in solving practical programming problems?
b. What will be the performance of a microprocessor with such a minimum set of instructions?
c. What facilities in a microprocessor are necessary to reduce the complexity and the system costs of a computer?
d. How to best utilize the current CMOS technology to build such MISC processors?

The MISC Instruction Set

What is the minimum set of instructions in a practical microprocessor? The CISC processors generally have 100 or more instructions. The RISC processors have about 50 instructions. In our investigations, it was obvious that 16 instructions are not sufficient to support all the necessary functions required in a microprocessor. 50 instructions are too many. The minimum number of instructions is somewhere between 16 and 32. A convenient choice is to limit the number of instructions to 32 and implement a microprocessor with five-bit instructions.

The instruction set implemented in MuP2 is shown in Figure One.

Figure One.

	MuP21 Instruction Set
Transfer Instructions:	`JUMP, CALL, RET, JZ, JCZ`
Memory Instructions:	`LOAD, STORE, LOADP, STOREP, LIT`
ALU Instructions:	`COM, XOR, AND, ADD, SHL, SHR, ADDNZ`
Register Instructions:	`LOADA, STOREA, DUP, DROP, OVER, NOP`

So far, we have implemented only 24 instructions, leaving some room for future expansion. This MISC instruction set seems to be adequate in the applications we have coded, including quite elaborate operating systems and demonstration programs.

It is interesting that we have an ADD instruction but not subtraction; that we have XOR but not OR; and that we have OVER but not SWAP. Obviously, subtraction can be synthesized by complement and addition. OR can be synthesized by complement, AND, and XOR. OVER and SWAP are very similar, in that they allow accessing the top of the data stack. However, it is difficult to determine which is more fundamental in a stack machine.

MuP21 Architecture

MuP21 is the first in a series of MISC microprocessors. The primary constraints on the design of this microprocessor were that it had to be housed in a 40-pin DIP package, and that the silicon die had to be less than 100 mils square. We determined that a 20-bit microprocessor could be implemented within these physical constraints. There would not be enough I/O pins to support a processor with wider data and address buses.

MuP21 must use DRAM as its primary memory, as DRAM offers the best bit density and the lowest cost per bit. However, it has to boot from ROM or other eight-bit memory devices, and it also has to address various I/O devices. Therefore, we need a memory coprocessor to handle the buses and to generate the proper control signals to the memory and I/O devices.

A very unique feature of MuP21 is to generate NTSC signals to drive a color TV monitor, because it will be targeted to many applications which use the TV monitor as the principal display device. A video coprocessor was designed to run in parallel with the main processor to display video frames stored in the main DRAM memory.

The main CPU in MuP21, thus, includes the following components:

a. a Return Stack to nest subroutine return addresses
b. a Data Stack to store parameters passing between subroutines
c. a T (Top) Register as the central holding register for operands
d. an ALU which takes operands from T and the top of Data Stack and returns the results of ALU operations to the T Register
e. an A (Address) Register to hold a memory address for fetching or storing data from/to memory
f. a PC (Program Counter) Register to hold the address of the next instruction, and
g. an Instruction Latch which holds four five-bit instructions to be executed in sequence.

The memory and data buses are 20-bits wide. The instructions are five-bits wide. Therefore, four instructions can be packed in each 20-bit word fetched from memory. This is a natural instruction pipeline. After four instructions are executed, the slower external memory is ready to supply the next set of four instructions. The processor can be four times faster than the memory. Fast cache memory and its associated control circuitry are not needed.

The execution speed of MuP21 is very fast because of the simple instruction set and the dual-stack architecture. The ALU instructions can be executed very fast because operands are taken from the T register and the top of the data stack, and the results are returned to the T register. There is no need to decode the source and destination registers. Actually, the ALU operates continuously. Once the data in the T register and the top of the data stack are stable, ALU results from COM (complement of T), SHL, SHR, XOR, AND, ADD, and conditional ADD are generated spontaneously. The ALU instruction only selects the proper results and gates them back into the T register. The operations of the MuP21 processor can thus be summarized in two steps:

a. Read a 20-bit word from memory and latch it into the instruction latch.
b. Execute the five-bit instructions by latching proper results into the T register.

MuP21 is, thus, much faster than RISC machines, because the RISC processor must follow the following sequence to execute one instruction:

a. Read an instruction from memory and latch it.
b. Decode the instruction and select the operand registers.
c. Execute the instruction.
d. Store results back into the selected destination register.

A stack-based processor is more advantageous than a register-based processor because the source and destination registers are defined in hardware and no register decoding is necessary.

MuP21 executes instructions at a speed of ten ns. per instruction. The peak execution rate is, thus, 100 MIPS. It achieves this remarkable performance using only the now-outdated, 1.2 micron CMOS process, because of the simplicity in its architecture and the MISC instruction set. Accessing the slower DRAM memory degrades its performance to about 80 MIPS.

Video Coprocessor

MuP21 has a video coprocessor which runs in parallel with the main CPU. The video coprocessor reads 20-bit words from the DRAM memory and interprets a 20-bit word as four five-bit instructions, similar to the main CPU. However, the video coprocessor instructions change the output voltage at the video output pin to generate an NTSC color video signal suitable for display on a standard TV monitor.

The video processor is synchronized to a 14.39 MHz external clock to maintain precise timing of the video output. Whenever it is ready to fetch a new word from the DRAM memory, it gets a word via the memory coprocessor without delay, because the video coprocessor has a higher priority over the main CPU, and the memory coprocessor will grant its memory request as soon as possible. After the video coprocessor gets a word from DRAM, it will execute four instructions before fetching the next word. During this interval, the main CPU can request memory access from the memory coprocessor. Hence, when the video coprocessor is turned on, it consumes 25% of the memory bandwidth of MuP21.

The instruction set of the video coprocessor is as follows:

Opcode	Hex	Name	Sot	Cycles
B	00	Black	x	1
S	17	Sync	x	1
R	1F	Refresh	2	1
K	13	Skip	0	1
C	15	Burst	x	1
P	0x	Pixel	x	1
J	18	Jump	0	0

When the MSB in a five-bit video instruction is set, the instruction causes special action in the video signal generator. When the MSB in an instruction is reset, the other four bits specify the color of one pixel to be displayed on the monitor. The assignments of bits are:

```
0 I G R B
```

where G, R, B stand for green, red, and blue, and I stands for intensity.

A video frame is first constructed in DRAM memory from the video instructions. When the video coprocessor is turned on (by setting the LSB in the Configuration Register), the video coprocessor fetches the instructions in sequence and executes them. The result is a continuous stream of analog signals at the video output pin. When this pin is connected to the input of a video monitor, color pictures will be shown on the monitor. The main processor can change the pixel instructions in the video frame to cause the picture to change dynamically.

Since the video frame is completely constructed in the DRAM memory, it is easy to produce video signals in either NTSC or PAL format. This feature makes MuP21 a very powerful and versatile device to produce TV images. It will, thus, find many applications where video output is needed.

Memory Coprocessor

The memory coprocessor in MuP21 is mostly hidden from the user. It performs the following tasks in the background:

a. It arbitrates DRAM access requests from the video coprocessor and the main CPU. The memory request from the video coprocessor has priority over that from the main CPU.

b. It generates the proper control signals to DRAM and SRAM memories, and also the I/O enable signal to I/O devices. A DRAM RAS cycle is 50 ns. SRAM and I/O have two accessing speeds: a slow cycle of 250 ns., and a fast cycle of 15 ns. The memory coprocessor allows MuP21 to use a variety of memory and I/O devices without additional interface circuitry.

c. It controls the address and data buses to the memory and I/O devices. When accessing DRAM memory, the 20-bit addresses are multiplexed over pins A0–A9, and the data bus consists of D0–D9 and AD10–AD19. When accessing SRAM memory during booting, the address bus consists of A0–A9 and AD10–AD19, while the eight-bit data bus is on D0–D7. When accessing I/O devices, the addresses are on A0–A9, and data are on D0–D9 and AD10–AD19.

Memory and I/O accesses are controlled by address lines and two bits in the Configuration Register. The memory maps of different memory and I/O devices are:

Address	Device
0–FFFFF	20-bit DRAM memory
12000–1203FF	slow 20-bit I/O devices
14000	Configuration Register
16000–1603FF	fast 20-bit I/O devices
18000–1BFFFF	fast eight-bit SRAM memory
1C000–1FFFFF	slow eight-bit SRAM memory

Internally, MuP21 maintains a 21-bit data/address bus. The MSB bit 20 is the carry bit in ALU operations. It also selects DRAM memory when low, and SRAM or I/O when high. According to the memory map, MuP21 addresses directly only 256 Kb of SRAM memory. However, bits 18–19 in the Configuration Register are forced on the address bus when reading or writing SRAM. This paging mechanism allows MuP21 to access 1 Mb of external SRAM memory.

Applications

MuP21 is a very powerful microprocessor because it is fast, and it has a fairly large addressing space. It also uses very little power. It is, therefore, suitable for a wide variety of applications in which high speed, low power consumption, and large addressing space are important factors in the design. Here is a list of potential applications for MuP21:

advanced video games
TV signage
video test-pattern generators
CAD design system
telephone switching system
handheld computers
high-speed communications systems
intelligent hard-disk controllers
robotics controllers

Conclusion

MuP21 is the first member of a family of microprocessors based on the MISC principles. It proves that there is still room

(Text concludes on page 10.)

(Fast Forthward, continued from page 46.)

THRU shall execute, while other words shall not. The remaining words shall be output to a file or other output stream so that a post-processor could process that text further.

A way to distinguish between those words that need to be executed and those words that can be appended to an output stream is required. A new word flag could be set for all Forth words that redirect the input stream. Suppose that it is named INPUT_SOURCE_SPECIFIER.

Typically, the input redirection words require input parameters such as (filename) strings or (block) numbers. To make sure those parameters will be available, a special class of constants might be necessary. The INPUT_SOURCE_SPECIFIER flag could be set for each of them to ensure that they will execute when encountered in the input stream.

Likewise, the preprocessing pass could strip any comments from the input stream. That requires the comment-introducing words to execute whenever they are encountered in the input stream. Therefore, we need to distinguish a new class of words using a different word flag, such as COMMENT_HERALD.

(As each of these new word flags is considered, do you see a well-classified Forth system taking shape, as I sometimes think I see?)

If the INPUT_SOURCE_SPECIFIER (and COMMENT_HERALD) word flag is checked by ?FIND, the DO_DEFINED vector can remain unchanged. That's because the only "found" words passed to the standard DO_DEFINED routine would be those we need to execute.

A new DO_UNDEFINED routine would have to be referenced in INTERPRET. It would merely write the input word to a selected output stream (or file).

Objects might be considered a different point of departure or a different means of travel. Objects alone do not dictate a destination.

More Stately Interpretation

To be able to leave the new interpreter state, another word needs to be able to be executed reliably. Suppose we give this word the name PREPROCESS_OFF, indicating that it switches away from the new preprocessor state back to a normal Forth state. To accommodate it, another word flag may be needed, such as PREPROCESS_STATE_SPECIFIER.

In keeping with the effort to make INTEPRET more flexible, the parameters required by ?FIND can be generated by execution vectors. The vectors could be placed in a lookup table, with enough slots for several more states.

The state could determine which vectors would be fetched from the lookup table to generate the word-sanctioning and word-rejecting parameters that ?FIND requires. To do so, consider using the following code to replace the snippet of code I offered in the last essay:

```
...
STATE @ TH-STATEVECTORS 2TOKEN@ EXECUTE
SWAP EXECUTE
( str-addr sanction-flags ignore-flags --)
?FIND
...
```

Wrap-up

Measures such as those Mitch and I have taken with respect to INTERPRET might be effective first steps along the path to implementing the visible and invisible portions of a module system, or the visible and invisible methods of an object system.

I just wish I knew what the next tune-up should be. I know it should involve objects or modules and that it must be compelling—probably by its ability to impart substantial kernel flexibility. Help, anybody?

An object-oriented Forth kernel might help the language's implementors exclusively. Objects might be considered a different point of departure or a different means of travel. Objects alone do not dictate a destination.

Implementors using objects are going to be as free as they ever were to produce a vanilla Forth.

(In that case, the implementation burden might have been eased, but the language taken up by the end user would still be plain old Forth. Accordingly, an object-based Forth kernel is not necessarily going to expose its object mechanisms to the end user. I would expect, however, that at least metacompilation will become much simpler, or disappear altogether.)

I still have my doubts about Forth's naturally occurring classes, however. Although perhaps not for the purposes of having true object classes or modules, Forth's words do seem to belong in clearly separated categories.

A large number of kernel Forth words can be nicely partitioned by various new word flags. The ANS Forth Quick Reference card categorizes a sizable set of words, so that Forth appears more approachable. For example, as long as you are not creating compiler extensions, you can ignore about 40 words. Through such categorizations, the attention of the novice can be narrowed to a much smaller, and much more relevant, set of words.

Classes or other categories of Forth words can be critical aids to learning Forth. This fact gives impetus to the effort to go ahead and build object orientation into Forth, as I am sure we will witness someday.

Note that it is one thing to admit the existence of categories. But it is an entirely different thing to argue that the words in those categories ought to be treated in an object-disciplined fashion. Perhaps word flags will proliferate where objects might have come forth.

Product Watch

NOVEMBER 1994

Laboratory Microsystems, Inc. announced a shareware version of WinForth™ that runs under Windows. The shareware package is the real thing, corresponding to LMI's WinForth Explorer retail package. You can use the product for evaluation purposes for up to 90 days. After that, you can purchase either the Explorer or Professional version of WinForth—or erase it. The only constraints on the shareware version are that the SAVE, TURNKEY, and MAKEDLL commands are disabled.

According to Ray Duncan, you can download the shareware version of WinForth from the LMI BBS at 310-306-3530. The BBS supports 2400, 9600, 14400, and 28800 baud; set your telecommunications parameters for eight bits, no parity, one stop bit. The file to download is WFSHR101.EXE; this is a self-extracting archive. Create a new directory named WINFORTH, copy WFSHR101.EXE into that directory, and run it. After the archive unpacks itself, see the READ.ME file for further instructions.

WinForth™ is not being placed in the public domain like Tom Zimmer's WIN32FORTH, which has a very similar name. E-mail any questions to Ray Duncan at lmi@cerf.net.

WinForth is a 16-bit segmented, direct-threaded-code (DTC) implementation that caches the top stack position. Among its features are a 286/287/387 assembler, a multi-window text file editor, trace and breakpoint utilities, a round-robin multi-tasker, and many programming examples. Windows API functions and DLLs can be called directly from WinForth.

DECEMBER 1994

The Saelig Company announced that a version of the TDS2020 Data Logger PC is now available with a resident ANS Forth kernel in 16K, leaving 45K for application and data. Extended memory can include up to 512K of battery-backed RAM. A 40 Mb hard disk is another option.

Although only 4" x 3" inches, the compact single-board controller includes features such as on-board, eight-channel, ten-bit A/D; and three-channel, eight-bit D/A converters; as well as bus and RS-232 interfaces. Library support is offered for stepper-motor control, interrupt handling, real-time multi-tasking, data logging, serial I/O, keyboard, and LCD.

JANUARY 1995

Forth, Inc. announced a database class library for its polyFORTH development system for PCs. The database library was previously available with the EXPRESS industrial control software package.

The database library is integrated with polyFORTH's GUI toolkit to facilitate development of OSF/Motif-like user interface screens, such as those for data entry.

Operators such as fetch, store, and display are overloaded for field classes such as byte strings and single- or double-precision numbers. A report generator is part of the new polyFORTH offering as well.

Files may reside in RAM for speed or on disk for permanence. Cached files offer the combined advantages of both types of files. Disk files are fully DOS-compatible.

COMPANIES MENTIONED

Laboratory Microsystems, Inc.
P.O. Box 10430
Marina del Rey, California 90295
Voice: 310-306-7412 Fax: 310-301-0761

FORTH, Inc.
111 N. Sepulveda Blvd.
Manhattan Beach, California 90266
Voice: 800-55-FORTH or 310-372-8493
Fax: 310-318-7130

The Saelig Company
1193 Moseley Road
Victor, New York 14564
Voice: 716-425-3753 Fax: 716-425-3835

Forth Wins in the Cellar

At the conclusion of its sixth annual design contest, *Circuit Cellar Ink* magazine reported last December that first place honors went to Eric Wilson and Gregg Norris. They are the developers of the Eye Mouse, a hardware and software project that used Forth. For this project, a mix of Forth and 68HC11 assembler was used.

The breakthrough for this project came when the two realized that, by detecting and processing microvolt-signals corresponding to the nerve signals that trigger eye orbit, eye motion could be used as the source for mouse input data. Taking advantage of existing ECG and EEG sensor technology, ionic current on the facial skin surface could be sensed, amplified, filtered, offset, and finally processed through a PIC1671.

Whenever the eye is moved, corresponding X and Y components of movement are detected. When the user's eyes have looked in one direction for half a second, the cursor moves in the corresponding direction until it is stopped by a double blink of the eyes. This interface to a computer could be a great benefit for people who must otherwise rely upon a second party to interpret yes and no eye blinks to express their needs and wishes.

A Forum for Exploring Forth Issues and Promoting Forth

Fast FORTHward

Can A Forth Kernel Use Objects?

Mike Elola
San Jose, California

In my last "Fast Forthward" essay, I suggested word flags to distinguish groups of kernel words, including inner interpreters and inline data handlers. The effectiveness of that approach was evidence that Forth has naturally occurring classes.

Such thinking tends to raise my expectations of developing a Forth kernel that is object oriented from the ground up. I long to know how much more refined Forth might become if it were tuned-up through modules or objects.

I showed you how word attributes such as `EXECUTE_IFF_AT_CF` and `HERALDS_INLINE_DATA` can be applied usefully to inner interpreters and inline data handlers, allowing programmer errors to be prevented.

If these two groups of words are good candidates for object classes, what are some of the possibly associated methods? Certainly methods such as "self-identify" or "skip-yourself" could be put to use. For example, a decompiler can be created that is able to print accurate information about any compiler-written memory locations.

...metacompilation will become much simpler, or it will disappear altogether.

I have already implemented such a Forth compiler and decompiler. But I did so without using an object system. I relied upon lookup tables instead. Those tables were entirely adequate for implementing polymorphic methods without all the bother of real classes.

The mantle of objects should produce more immediate benefits. Objects should produce a substantially better programming environment, perhaps by eliminating metacompilation through enhanced kernel flexibility.

Misplaced Models

Perhaps the problem is that objects are best used to model the behavior of things that have a separate existence in reality. At the kernel level, we don't have real objects to be modeled. Furthermore, we want the Forth runtime to remain streamlined.

Nevertheless, library functions such as I/O functions are presumably benefiting from object orientation within other languages, such as C++.

To gain the advantages of objects or modules, we perhaps need to be programming at less-primitive kernel levels. For example, we may need to be programming at a context where dynamic memory management or other broadly applicable services are needed.

Some applications just don't need these flourishes, however. Others may need sophisticated versions of features such as memory management. Perhaps modules or objects could help manage scalable solutions for such features, so that the need for metacompilation can be eliminated.

To explore how a more flexible kernel might be created, let's consider the Forth text interpreter. Different classes of objects could assume responsibility for the different roles of the text interpreter.

One object could be responsible for managing the input stream and detecting its exhaustion. It could communicate with other objects that initialize the interpreter, finalize its operation for a given input run, and change its internal state.

Although what follows is not an object treatment of the text interpreter, it offers flexibility that is in the same spirit as that in a system of objects. (Vectors can be considered roughly analogous to the methods of object implementations.)

A Three-Vector INTERPRET

A few years ago, Mitch Bradley (long-time proponent of Open Firmware) reported to those of us at the Silicon Valley FIG Chapter meeting about his use of a text interpreter framework. Mitch gleefully described how the incorporation of three execution vectors in `INTERPRET` gave him a very flexible system of text interpretation.

His execution vectors were named `DO_UNDEFINED`, `DO_DEFINED`, and `DO_LITERAL`. These vectors permit the expansion of the text interpreter in specific behavioral areas: a changeable "word not found" error-handling behavior, a changeable literal-handling behavior, and a changeable compiling or interpreting behavior. By referencing differently behaved routines for each of these vectors, he found many useful combinations of routines.

A Forth Preprocessor

To use Forth's text interpreter as a preprocessor, we could add a new interpreter state. In that state, the text interpreter would traverse the input stream, following any twists and turns brought about by words like `LOAD` and `THRU` (and equivalent file operations).

Within the new interpreter state, words like `LOAD` and

(Continues on page 44.)

Correction to the ANSI Standard Quick Reference

Corrections for WITHIN and >R

The description of `WITHIN` that appears in the ANSI Standard Forth Quick Reference card that *FD* readers received from FIG is erroneous.

Although L. Greg Lisle informed me about an error way back in September, I have neglected mentioning it until now. Ostensibly, the extra time was needed to come up with a clearer explanation of the ANSI Forth version of `WITHIN`. After missing the point several times in ensuing discussions with colleagues, I will try my hand once again.

Without a doubt, the ANSI Forth version of this word is more difficult to comprehend than previous versions. John Rible, X3J14 committee member and scribe, explained to me that the standard avoids reference to the two's complement number system, even though its mention could help readers to understand the behavior of `WITHIN`. In any case, John put me on the right track with the notion of circular systems of number representation.

Spatially, the notion of "betweenness" is an analog of the ANSI Forth `WITHIN`. Let's consider the subject of world-wide travel to help us understand `WITHIN`.

If the world were flat, Hawaii would always be between between Japan and California—regardless of the direction of travel you choose. But considering that the world is round, you could take the long way around the globe and really mess up an otherwise simple concept of what "in-between" means. (Travel advisory: slippery discussion thread ahead.)

Suppose only a westerly travel direction is permitted. Then, Hawaii is not encountered first if Japan is the point of departure. So, according to this travel restriction and the departure (Japan) and arrival (California) locales, Hawaii is not between Japan and California.

However, you could say that Japan is between Hawaii and California, because you would encounter Japan first if you left Hawaii heading west.

Unless we restrict ourselves to either a westerly or an easterly direction of travel, everything can be said to be between everything else on a spherical surface.

To understand the operation of `WITHIN`, it is essential to know the direction of the sense of comparisons. Only one sense of direction is permissible in order for `WITHIN` to be helpful. The analogy of unidirectional travel around the globe is, therefore, an appropriate one.

The input portion of the stack diagram for `WITHIN` is given (in reference-card style) by:

```
n1  n2  n3 --
```

where n1 is the number whose centrality is in question, n2 is the lower limit, and n3 is the upper limit. To account for the circularity of certain computer numbering systems, the ANSI standard offers these expressions to describe when `WITHIN` returns true:

```
( n2<n3 and ( n2<=n1 and n1<n3 ) ) or
( n2>n3 and ( n2<=n1 or n1<n3 ) )
```

Because n2 is described as the lower limit and n3 is described as the upper limit, the second expression seems strangely contradictory. This was one of the stumbling blocks that fouled my process of learning and acceptance. (Perhaps the madness lies in the labeling of parameters as lower and upper limits. Perhaps boundary-start and boundary-end are better label choices.)

Based on his acquaintance with Pygmy Forth, L. Greg Lisle recognized that these three different expressions also describe when `WITHIN` returns true:

```
(1)     n2 > n3 > n1
(2)     n1 >= n2 > n3
(3)     n2 <= n1 < n3
```

I'll even hazard (very unwisely) to say a bit more. Expressions (1) and (2) both deal with two halves of the same path that is being established by the backwards relationship of n2 (lower limit) and n3 (upper limit). Further, the solution set (or path) they define (due to the direction they establish) is likely to be deceptively large. In integers, the solution sets for expressions (1) and (2) are:

```
(1)     n3-1, n3-2, n3-3, ... -∞
(2)     n2, n2+1, n2+2, ... +∞
```

Back to the globe analogy, n2 > n3 selects the equivalent of the circuitous, or long-way-to-home path—despite the possible proximity of the departure and arrival points.

It's as if such a path can't be described in one equation because of an irregularity at some point on the globe (such as the international date line). That irregularity introduces a phantom boundary that cannot be crossed as part of one continuous path. The barrier corresponds to the point where the largest positive integer segues into the smallest negative integer—due to the setting of the sign bit and the clearing of lower-order bits—when it is incremented by one on most Forth systems (an overflow condition for signed numbers).

Rhetoric aside, here is what you should do to correct your ANSI Forth reference card: Strike out the current description of `WITHIN` on the card. It's in the "Comparison Operations" section. Replace it with either the standard's two elaborate expressions, or L. Greg Lisle's simpler trio of expressions.

(I know! You could each vote for the version you prefer to see on the card. I tend to like the simple trio as opposed to the elaborate duo—but that's just my two cents.)

Please continue to use this forum to offer clarifications, or even corrections to my corrections. (Send your e-mail to elolam@aol.com.) I thank L. Greg Lisle and John Rible for their patience and their help.

But before we leave the subject of corrections, L. Greg Lisle also pointed out long ago that the word `>R` is missing from the quick reference card. Please add it to your card in the section entitled "Manipulating Stack Items."

First in a new series of books by the Forth Interest Group

Thinking FORTH

Business, industry, and education are discovering that FORTH is an especially effective language for producing compact, efficient applications for realtime, real-world tasks. And now there's Thinking Forth—an instructive guide that illustrates the *elegant logic* behind the language and shows how to apply specific problem-solving tools to software, regardless of your programming environment.

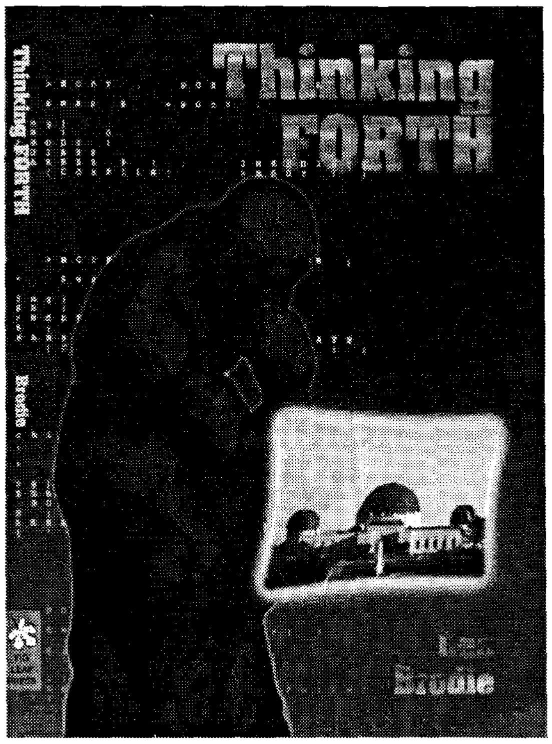
It combines the philosophy behind Forth with traditional, disciplined approaches to software development— to give you a basis for writing more readable, easier-to-write, and easier-to-maintain software applications in any language.

Written in the same lucid, humorous style as the author's *Starting Forth* and packed with detailed coding samples aand illustrations, Thinking Forth reviews fundamental Forth concepts and takes you from the initial specification of your software project through the analysis and implementation process, showing how to simplify your program and still keep it flexible throughout. Both beginning and experienced programmers will gain a better understanding and mastery of such topics as

- FORTH style and conventions
- decomposition
- factoring
- handling data
- simplifying control structures
- and more.

And, to give you an idea of how these concepts can be applied, Thinking Forth contains revealing interviews with real-life users and with Forth's creator, Charles H. Moore.

To program intelligently, you must first *think* intelligently, and that's where Thinking Forth comes in.



Available Now

$18

Includes Member Discount

TO PROGRAM INTELLIGENTLY, YOU MUST FIRST THINK INTELLIGENTLY